MIT/LCS/TR-370

A COMPILER FOR THE MIT TAGGED-TOKEN DATAFLOW ARCHITECTURE

Kenneth R. Traub

August 1986

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Compiler for the MIT Tagged-token Dataflow Architecture

August 1986

Kenneth R. Traub

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfilment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Compiler for the MIT Tagged-Token Dataflow Architecture

by

Kenneth R. Traub

Submitted to the Department of Electrical Engineering and Computer Science on August 8, 1986, in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

Abstract

Compilation of the programming language Id Nouveau into machine code for the MIT tagged-token dataflow architecture is thoroughly described. Id Nouveau is a higher-order functional language augmented with a novel data structure facility known as I-Structures. The tagged-token dataflow architecture is a dataflow computer of the dynamic variety.

Compilation takes place in two steps. In the first step, the Id Nouveau program is converted into an abstract dataflow graph called a program graph. Program graphs embody no detailed knowledge of the target architecture, yet have a very precise operational semantics. At the same time, they represent data and control flow in a way very convenient for program transformation. Several common optimizing transformations are discussed.

The second step of compilation converts the program graph into machine code for the tagged-token architecture, taking into account the machine's finite resources. Peephole optimizations for machine code are discussed, and a general-purpose optimization algorithm is given.

Thesis Supervisor: Arvind
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Compilers, Data Flow, Id, I-Structures, Optimizing Compilers, Tagged Token Dataflow Architecture

A Compiler for the MIT Tagged-Token Dataflow Architecture

by

Kenneth R. Traub

Submitted to the Department of Electrical Engineering and Computer Science on August 8, 1986, in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

Abstract

Compilation of the programming language Id Nouveau into machine code for the MIT tagged-token dataflow architecture is thoroughly described. Id Nouveau is a higher-order functional language augmented with a novel data structure facility known as I-Structures. The tagged-token dataflow architecture is a dataflow computer of the dynamic variety.

Compilation takes place in two steps. In the first step, the Id Nouveau program is converted into an abstract dataflow graph called a program graph. Program graphs embody no detailed knowledge of the target architecture, yet have a very precise operational semantics. At the same time, they represent data and control flow in a way very convenient for program transformation. Several common optimizing transformations are discussed.

The second step of compilation converts the program graph into machine code for the tagged-token architecture, taking into account the machine's finite resources. Peephole optimizations for machine code are discussed, and a general-purpose optimization algorithm is given.

Thesis Supervisor: Arvind
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Compilers, Data Flow, Id, I-Structures, Optimizing Compilers, Tagged Token Dataflow Architecture

Acknowledgments

The timely completion of this thesis would not have been possible without the generous cooperation of my advisor, Professor Arvind. I am grateful for his enthusiasm, confidence in my ability, and guidance.

I have been privileged to be a part of the Computation Structures Group, whose talented members have provided a stimulating and exciting environment in which to work. In particular, I wish to thank Steve Heller, Andrew Chien, and David Culler for their moral support, as well as Greg Papadopoulos, Richard Soley, Mike Beckerle, and Bob Iannucci for their enthusiasm for the compiler project.

Others have made valuable contributions to the compiler. Many of the concepts were derived from Vinod Kathail's original Id Compiler. The early organizational decisions were made in conjunction with Steve Heller. Much helpful feedback was gained from the first users of the old compiler, especially Gino Maa, Steve Brobst, and Kattamuri Ekanadham. David Culler provided the impetus for considering peephole optimizations. Serge Plotkin validated the common subexpression elimination algorithm. Mike Beckerle first brought attribute grammars to my attention.

The quality of the thesis was greatly improved thanks to two souls kind and brave enough to plow through early drafts, Steve Heller and Professor Nikhil.

Finally, none of it would have been possible without the unwavering love and support of my parents. To them I give my greatest thanks.

The author is supported in part by a grant from the National Science Foundation.

Acknowledgments

The timely completion of this thesis would not have been possible without the generous cooperation of my adviser, Professor Arvind. I am grateful for his enthusiasm, confidence in my ability, and guidance.

For Julie

I have been privileged to be a part of the Computation Structures Group, whose talented members have provided a stimulating and exciting environment in which to work. In particular, I wish to thank Steve Heller, Andrew Chien, and David Culler for their moral support, as well as Greg Papadopoulos, Richard Soley, Mike Beckerle, and Bob Iannucci for their enthusiasm for the compiler project.

Others have made valuable contributions to the compiler. Many of the concepts were derived from Vinod Kathail's original Id Compiler. The early organizational decisions were made in conjunction with Steve Heller. Much helpful feedback was gained from the first users of the old compiler, especially Gino Maa, Steve Brobst, and Kattamuri Ekanadham. David Culler provided the impetus for considering peephole optimizations. Serge Plotkin validated the common subexpression elimination algorithm. Mike Beckerle first brought attribute grammars to my attention.

The quality of the thesis was greatly improved thanks to two souls kind and brave enough to plow through early drafts, Steve Heller and Professor Nikhil.

Finally, none of it would have been possible without the unwavering love and support of my parents. To them I give my greatest thanks.

The author is supported in part by a grant from the National Science Foundation.

Table of Contents

Table of Contents

1. Introduction

The search for a scalable parallel computer architecture has led the Computation Structures Group of MIT's Laboratory for Computer Science into developments in two areas. On one front, a general purpose programming language suitable for execution on parallel architectures has been developed; its present name is Id Nouveau. On the other front, a dataflow architecture called the Tagged-Token Dataflow Architecture, or TTDA for short, has been designed and extensively simulated. Id Nouveau can be described as a language combining the features of *functional*, or *applicative*, languages with a novel data structure facility called *I-Structures*. The TTDA falls into the general category of *dynamic* dataflow architectures, other examples of which include the Manchester Machine [Gurd 85] and ETL's Sigma-1 [Hiraki 84].

As always, the missing link between language and machine is the compiler. In this thesis we examine how functional *cum* I-structure languages can be translated into object code for dynamic dataflow architectures. While our discussion is based on Id Nouveau and the TTDA, most of what is said is applicable to other dataflow languages and architectures. Two requirements must be met: the dataflow graphs must correctly implement the semantics of the language, and the graphs must conform to any constraints and peculiarities of the target architecture. Tied intimately with the latter is a third requirement that the resources required for execution of the program, of which only a finite quantity are available, be controlled in certain ways.

The outline of the thesis is as follows. In Chapter 2, we briefly describe a minimal subset of Id Nouveau called Id Kernel. Id Kernel is in not a "toy" language, however; every construct in Id Nouveau has a counterpart in Id Kernel, though perhaps with small syntactic modifications. A full description of Id Nouveau can be found in [Nikhil 86], where an algorithm for conversion of Id Nouveau programs into Id Kernel may also be found.

Chapter 3 begins the compilation process, by describing how an Id Kernel program is converted into an abstract sort of dataflow graph which we term the *program graph*. There is an approximately one-to-one correspondence between constructs in the source program and instructions in the program graph, even though it may take many machine code instructions to implement a particular construct. The program graph is therefore not suitable for direct execution on any dataflow architecture, although it bears a strong resemblance to the dataflow graphs used by a wide variety of dataflow machines. In fact, the program graph is much closer in spirit to the "data flow graphs" used by optimizing compilers for conventional languages. One important difference is that a conventional compiler uses data flow graphs as an auxiliary data structure to aid analysis, but for us the program graph is the *only* intermediate data structure of any importance, and it captures everything we might want to know about the source program. Consequently, it is possible to give a rather precise operational interpretation of the program graph.

The similarity between program graphs and the dataflow graphs used by conventional compilers leads naturally into Chapter 4, where we examine optimizing transformations that can be performed on the program graph. As it turns out, most if not all of the common optimizations employed by conventional compilers have counterparts in the dataflow domain. In fact, the elegance of a dataflow language combined with the program graph representation allows many of these optimizations to be performed with greater ease, effectiveness, and confidence.

The program graph is but an intermediate step on the way to code generation for a dataflow machine. In Chapter 5 we examine the architecture of one such machine, the MIT Tagged-Token Dataflow Architecture. While we do not attempt to give a description complete to the last NAND gate, we present enough detail to gain an understanding of the constraints on the instruction set, and how the finite nature of the machine will influence the translation of a programming language in which resource management is totally the responsibility of the compiler and machine.

One requirement that falls out of consideration of the machine is the need for additional arcs in the dataflow graph called *triggers* and *signals*. In Chapter 6 an algorithm for their introduction is presented; the resulting graph is called a *Well-Connected* program graph.

With the Tagged-Token Architecture in hand, Chapter 7 presents the translation from Well-Connected program graph into machine graph, or object code for the TTDA. Because the program graph already embraces the notion of data flow, translation from program graph to machine graph is even more straightforward than from Id Kernel to program graph. In fact, translating to machine graph simply involves the context-free substitution of machine code for program graph constructs. The subtlety arises in the definitions of the substitutions, for they must heed the restrictions imposed by the architecture while still implementing the full generality of the program graph.

Peephole optimization is commonly employed by conventional compilers to improve code generated from straightforward translation of programs. Dataflow compilers are no exception, and in Chapter 8 we see how peephole optimization applies to dataflow code. A general purpose, pattern driven optimization algorithm is presented. To the author's knowledge, this is the first discussion of peephole optimization as it relates to dataflow graphs.

Finally, we conclude in Chapter 9 with a summary, a look at the present state of implementation, and some directions for future research.

machine graph is even more straightforward than from Id Kernel to program graph. In fact, translating to machine graph simply involves the context-free substitution of machine code for program graph constructs. The subtlety arises in the definitions of the substitutions, for they must heed the restrictions imposed by the architecture while still implementing the full generality of the program graph.

Peephole optimization is commonly employed by conventional compilers to improve code generated from straightforward translation of programs. Dataflow compilers are no exception, and in Chapter 8 we see how peephole optimization applies to dataflow code. A general purpose, pattern driven optimization algorithm is presented. To the author's knowledge, this is the first discussion of peephole optimization as it relates to dataflow graphs.

Finally, we conclude in Chapter 9 with a summary, a look at the present state of implementation, and some directions for future research.

2. Id Nouveau

Id Nouveau [Nikhil 86] is a programming language developed by the Computation Structures Group at MIT's Laboratory for Computer Science. An evolutionary successor to the language Id [Arvind 78], Id Nouveau can be described as a functional language enhanced with a data structuring facility known as I-Structures [Arvind 86a, Nikhil 86]. With its functional core, Id Nouveau is primarily concerned with expressions and abstractions of expressions (functions). Functions are first-class objects in Id Nouveau, and may be passed to and returned from other functions, stored in data structures, *etc.*. I-Structures are a novel sort of data structure which lay midway in power between purely functional data structures and storage as it is found in imperative languages. More will be said about them momentarily.

We do not attempt to give a complete description of Id Nouveau here, as this has admirably been done in [Nikhil 86]. In fact, throughout the thesis we will restrict our attention to a subset of Id Nouveau, referred to as Id Kernel. Id Kernel has the full expressive power of Id Nouveau; it lacks only those constructs which can be syntactically converted to other constructs that are retained in Id Kernel.

2.1 Expressions, *if*, and *let*

The simplest Id programs[1] are just expressions:

```
6.847                    1 + 1                    3*5 + 4^(78/6.023E-23)
```

Id possesses the usual complement of arithmetic, relational, and logical operators over the usual integers, floats, and booleans. A conditional, or `if`, expression is also allowed:

```
if 1 + 1 = 2 then 123 else 321     (if 1+1=2 then 123 else 321) * 34
```

Since the conditional is an expression, rather than a statement, it must always have both a `then` part and an `else` part. If one of these were missing, the expression would not always have a value.

[1] Throughout this thesis we will use the names "Id Nouveau" and "Id" interchangeably. Unless specifically stated, we are never referring to the original language Id as described in [Arvind 78].

Names may be associated with expressions through use of the `let` construct:

```
let
  a = 5 * 3;
  c = 2 + 1
in
  (a + c) / (a - c)
```

The value of a `let` expression is the value of the expression following the `in` keyword, `1.5` in the example above. The variables defined in a `let` have a scope which extends over the entire `let` expression, not just over the `in` expression[2]. A consequence of this rule is that the bindings may appear in any order, and therefore the semicolon serves no sequencing role as it does in imperative languages. The following expression evaluates to `12`.

```
let
  a = if c = 2 then 10 else 3521;
  c = (let a = 5 * 3 in a * a) - 223
in
  a + c
```

This example also illustrates the lexical scoping of identifiers; the `a` defined in the inner `let` is a completely different variable from the `a` defined in the outer `let`. We can always systematically rename identifiers so that no identifier appears on the left hand side of a binding more than once. If we had done so for the previous example, we would have:

```
let
  a = if c = 2 then 10 else 3521;
  c = (let b = 5 * 3 in b * b) - 223
in
  a + c
```

We say an occurrence of an identifier is *free* within an expression if the expression does not also contain the binding assigning that identifier its value. Hence, the right hand side of the first binding above has one free variable, `c`, the right hand side of the second has no free variables, and the expression following the keyword `in` has two free variables, `a` and `c`. The entire expression has no free variables.

Let us emphasize that bindings in Id are not at all like assignment statements in

[2] This scoping rule is sometimes referred to as "letrec" scoping.

imperative languages, despite the syntactic similarity. An Id identifier is simply a name given to the value of an expression so that it may be used in several places. Consequently, the same identifier may not appear on the left hand side of more than one binding in a `let`, since each identifier must map to exactly one value. This is often called the "single assignment rule", although somewhat of a misnomer since a binding does not necessarily entail assignment in the sense of filling in a storage location set aside for a variable. In many dataflow implementations, in fact, there are no fixed locations corresponding to identifiers.

2.2 Definition and Application

Id Nouveau allows us to abstract over expressions, yielding functions. Functions are defined with the keyword `def`

```
def mean2 x y = (x + y) / 2;
```

and invoked by juxtaposing the function name with its arguments:

```
mean2 30 40
```

The expression above returns `35`. Id functions are "curried", which means that they can take their arguments one at a time. Therefore, `mean2 0` is a perfectly valid Id expression, one that returns a function which halves its argument. The number of arguments to which a function must be applied before its definition is evaluated is called the function's *arity*; `mean2` has an arity of 2. The arity of a function is determined syntactically as the number of identifiers (formal parameters) that follow the function name on the left hand side of the `def` that defines it.

All definitions in Id Kernel are *closed*, meaning that the free variables of the right hand side of a definition must be a subset of the formals. The full Id Nouveau language permits definitions, not necessarily closed, within `let` expressions. These can be converted to closed Id Kernel definitions through *Lambda Lifting* [Johnsson 85].

2.3 I-Structures

The features of Id Nouveau described so far are the same as those found in *functional* languages. The term "functional" arises because the value returned by a procedure application depends only on its arguments; two calls to the same procedure with the same arguments are guaranteed to return indistinguishable results. This fact accounts for several desirable properties of functional languages. Functional languages possess *referential transparency*, or the ability to have the expression defining a variable be substituted for occurrences of that variable without changing the meaning of the program. They also possess the Church-Rosser property, also called the confluence property, which guarantees that the answer computed is unaffected by the choice of which subexpressions to evaluate first (although some orders of evaluation may fail to produce an answer at all). This property is what interests architects of parallel machines, for it insures overall program determinacy even if the machine exhibits non-determinacy in instruction scheduling. Of course, we cannot hope to evaluate a + 3 if we do not have a value for a — the order of evaluation is still constrained by *data dependencies*.

Data structures in functional languages are created by invoking a constructor function, which takes as arguments the values with which to fill the components of the structure. A familiar example of a functional constructor is Lisp's cons. Referential transparency demands that the values returned by two calls to the same constructor function with the same arguments must never be distinguishable. Thus, in a functional language, we can never alter a data structure once it has been created, and consequently we must specify the contents of all elements of the structure at creation time. Some latitude exists as to *how* we specify the arguments; in particular, it is sufficient to supply an expression computing each component without actually waiting for their values to have been computed. In that case, an attempt to extract the value of a component will be suspended until the component's expression has completed evaluation. This sort of functional data structure is called *non-strict*, or *lenient*. A variation on this is to prevent any evaluation of the expression computing a component's value until it is certain that value is needed, *i.e.*, until the first fetch of that component is attempted. This kind of structure is called *lazy*, or *demand-driven*. Because these various kinds of functional data structures differ only in the ordering of subexpression evaluation, the Church-Rosser property implies that they are semantically equivalent (modulo termination properties).

I-Structures, on the other hand, are not semantically equivalent to functional data structures; in fact, a language that includes them is no longer functional. An I-Structure is a one-dimensional array, or vector, which is completely empty when constructed. In Id Nouveau, the expression

```
array(1..10)
```

returns an empty I-structure whose elements are numbered consecutively 1 through 10. An I-structure element may be filled in using an I-Structure Store construct within a `let` expression:

```
let
  a = array(1..3);
  a[1] = 25;
  a[2] = 6;
  a[3] = 4
in
  a
```

This program creates and returns an I-structure whose three elements receive the values 25, 6, and 4. There is no restriction on where an I-structure Store may appear in relation to the `array` expression that constructs it. Thus, we can write the following procedure, which initializes the first and last element of the I-structure it receives as argument.

```
def fill_ends a =
  let
    lower = lower_bound a;
    upper = upper_bound a;
    a[lower] = 0;
    a[upper] = 0
  in
    ();
```

Notice that this procedure does not return any values, it simply has the side-effect of filling in some slots of its argument. We might use this procedure in the following way:

```
def make_thing n =
  let
    s = array(1..n);
    = fill_ends s;
  in
    s;
```

Notice that `fill_ends` is invoked by placing it on the right hand side of an equal sign. This syntax is used to cause the evaluation of any expression solely for its side effects, and is called a "command" rather than a binding.

I-structures certainly sacrifice the language's referential transparency property, since two calls to `array(1..10)` return two different I-structures that may be filled in differently. That is, the following two programs are not equivalent:

```
def prog1 n =
  let
    b = array(1..n);
    c = array(1..n)
  in
    ...;

def prog2 n =
  let
    a = array(1..n);
    b = a;
    c = a
  in
    ...;
```

In `prog1`, filling in an element of `b` cannot affect any computation using `c`, while in `prog2`, filling in an element of `b` does affect computations using `c`, since `b` and `c` refer to the same structure.

We can accept a lack of referential transparency, but because we are interested in parallel implementations we still want to retain the Church-Rosser property, or the ability to retain program determinacy without unnecessarily constraining order of evaluation. For this reason, I-Structures have two restrictions. First, an attempt to fetch an empty I-Structure element becomes suspended until that element is filled in by an I-structure Store, at which time the fetch returns the newly stored value. It is therefore impossible tell whether an I-structure element is empty or not. Second, an attempt to write an individual I-structure element more than once signals an error condition which voids the result of the entire program. No matter how we interleave execution of reads and writes, therefore, every fetch to a given I-structure element always returns the same value, namely, the value written into that element by the single I-structure Store for that element. Determinacy has been preserved.

In Id Nouveau, the usual array syntax denotes I-structure fetches:

```
def sum_ends a =
  let
    lower = lower_bound a;
    upper = upper_bound a
  in
    a[lower] + a[upper];
```

Id Nouveau with its I-structures represents a new class of languages having many of the desirable features of purely functional languages while offering potentially greater efficiency in data structure manipulation. This is especially true in scientific computation, where functional data structures often imply a great deal of copying, with a concomitant loss in available parallelism. A typical example is a program in which the boundaries of a large matrix are initialized, and then the inner elements progressively computed from the outer ones. Using I-structures, only one matrix need ever be allocated.

I-structure languages are in their infancy, and their formal properties are still under investigation. The most recent addition to our understanding of them is in the operational semantics based on rewriting given for Id Nouveau in [Nikhil 86]. Their translation into dataflow graphs as given in this thesis is another form of operational semantics, albeit of a far less abstract sort. A denotational semantics of Id Nouveau is under development [Pingali 86]. One difficulty in formulating the mathematical semantics is that I-structure languages are *non-sequential*, in the sense that it is not possible to give at compile time a total ordering on the evaluation of subexpressions. A discussion of this point is outside the scope of this thesis.

2.4 Loops

The constructs described in the previous sections comprise the complete Id Kernel subset of Id Nouveau, possessing the full expressive power of the latter language. In this thesis, however, we shall retain one additional Id Nouveau feature in Id Kernel: the loop construct. The `while` form of an Id Nouveau loop is illustrated below:

```
def fill_it a =
  let
    i = lower_bound a;
    sum = 0
  in
    while i <= upper_bound a do
      val = (upper_bound a - lower_bound a) ^ 2 - i * i;
      a[i] = val;
      new sum = sum + val;
      new i = i + 1
    return sum
```

This program fills each element of its argument array with a value computed from the element's subscript, and returns the sum of the all elements. The body of the loop resembles the body of a `let`: it contains ordinary bindings (like that for `val`) which name subcomputations, I-structure stores which fill in I-structure elements, and although not shown, it can also contain commands which specify computations to be performed for side effect. Loop bodies can also contain *new bindings*, like that for `sum` and `i` above; identifiers appearing on the left hand sides of new bindings are called *newified variables* (for lack of a better name). The first time the body is evaluated, the newified variables assume the values they had outside the loop, *e.g.*, `0` for `sum` in the example. The new bindings describe how to compute the values the newified variables take on for the next iteration for the loop; in the example, therefore, `i` is incremented each time through the loop. Finally, the expression following the keyword `return` is evaluated in a context wherein the newified variables have the values computed during the last iteration of the loop.

Two points worthy of mention: the new bindings only affect the values of the newified variables for appearances within the loop expression. For example, if there was a reference to the variable `sum` appearing outside the loop, it would have the value zero. Second, while an ordinary binding within a loop introduces a new variable name, a new binding must refer to a variable already introduced outside the loop, otherwise it would have no value for the first iteration of the loop. One way to understand a loop as is an alternative syntactic formulation of a tail-recursive procedure having one formal for each newified variable:

```
def fill_it a =
  let
    fill_it_iter i sum =
      if i <= upper_bound a then
        let
          val = (upper_bound a - lower_bound a) ^ 2 - i * i;
          a[i] = val
        in
          fill_it_iter (i + 1) (sum + val)
      else
        sum;
  in
    fill_it_iter (lower_bound a) 0;
```

(In Id Kernel, the internal definition of `fill_it_iter` would have to be lifted out via lambda-lifting to top level, resulting in the addition of a formal parameter for a.) Thus, loops are not an essential feature of Id Nouveau, since they can be mechanically translated into tail-recursion (an algorithm for this is given in [Nikhil 86]). Nevertheless, we will retain loops in this thesis, and compile them differently from their tail-recursive equivalents. Our motivations are twofold. First, explicit representation of loops in the abstract program graph will reveal opportunities for a wealth of optimizing transformations, such as hoisting loop invariants. Second, the dataflow architecture provides mechanisms for executing loops more efficiently than recursive procedures.

One caveat is in order: the dataflow translation of loops we give will have slightly different termination properties than for the translation of their tail-recursive equivalents. That is, it is possible to write a loop program such that its translation as a loop will deadlock while the corresponding tail-recursive translation will run to completion. This is a direct consequence of resource constraints imposed on loops to maintain their efficiency. We note here that because it is possible for a compiler to translate between tail-recursive and loop formulations, the choice of execution mechanism is decoupled from the programmer's choice of syntactic representation. The compiler is free to translate potentially deadlocking loops into tail recursion, although we will not discuss methods of determining when this is necessary or desirable.

```
def fill_it a =
  let
    fill_it_iter i sum =
      if i <= upper_bound a then
          let
            val = (upper_bound a - lower_bound a) ^ 2 - i * i;
            a[i] = val
          in
            fill_it_iter (i + 1) (sum + val)
      else
          sum;
  in
    fill_it_iter (lower_bound a) 0;
```

(In Id Kernel, the internal definition of `fill_it_iter` would have to be lifted out via lambda-lifting to top level, resulting in the addition of a formal parameter for a.) Thus, loops are not an essential feature of Id Nouveau, since they can be mechanically translated into tail-recursion (an algorithm for this is given in [Nikhil 85]). Nevertheless, we will retain loops in this thesis, and compile them differently from their tail-recursive equivalents. Our motivations are twofold. First, explicit representation of loops in the abstract program graph will reveal opportunities for a wealth of optimizing transformations, such as hoisting loop invariants. Second, the dataflow architecture provides mechanisms for executing loops more efficiently than recursive procedures.

One caveat is in order: the dataflow translation of loops we give will have slightly different termination properties than for the translation of their tail-recursive equivalents. That is, it is possible to write a loop program such that its translation as a loop will deadlock while the corresponding tail-recursive translation will run to completion. This is a direct consequence of resource constraints imposed on loops to maintain their efficiency. We note here that because it is possible for a compiler to translate between tail-recursive and loop formulations, the choice of execution mechanism is decoupled from the programmer's choice of syntactic representation. The compiler is free to translate potentially deadlocking loops into tail recursion, although we will not discuss methods of determining when this is necessary or desirable.

3. The Program Graph

The program graph is composed of a collection of *instructions*, each with an *opcode* that identifies its function, and some number of *inputs* and *outputs*. A directed *arc* connecting an output to an input allows a piece of data called a *token* to flow along that arc. For convenience, we will assume that any particular output may be connected to an arbitrary number of inputs (but not the reverse); the token emitted by an output is copied and sent to all inputs to which the output is connected.

A *firing rule* describes the behavior of instructions with a given opcode. The firing rule explains, for each output, when a token may be emitted from that output, and how the value on that token is to be computed. For some instructions, firing rules will also indicate side-effects to be brought about. The rules will generally be in terms of the tokens arriving on the instruction's inputs. For example, the familiar + instruction is defined by the following firing rule:

——— **Output** *Output* **of** + ———

Pre-Condition: *Input*$_1$ present and *Input*$_2$ present.

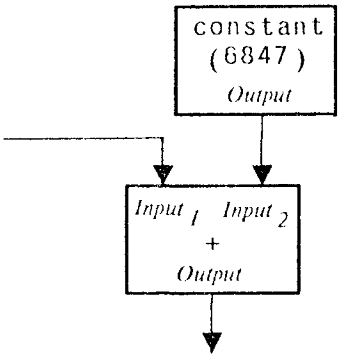
Value Produced: *Input*$_1$ + *Input*$_2$

Program graph instructions differ from machine graph instructions in their complexity. A single program graph instruction may have many inputs and outputs, and the firing rule may stipulate that some outputs produce tokens even though no tokens arrive on some inputs. These powerful instructions provide a concise framework for expressing the dataflow translation of Id Nouveau programs without considering details of the tagged-token dataflow architecture.

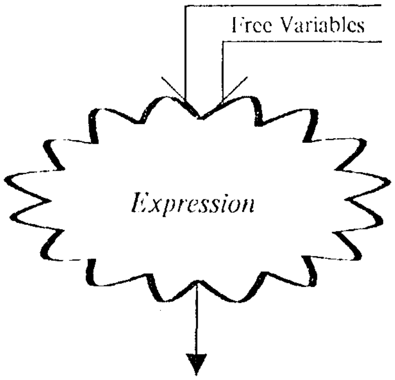
The firing rules will clearly define the behavior of instructions when presented with no more than one token on each of their inputs. On the other hand, we will be using instructions in situations where they can conceivably receive several tokens on the same input, such as within the body of a loop. For now, we shall not concern ourselves with rules for distinguishing among multiple sets of tokens, relying instead on the reader's intuition. Explicit attention to proper matching will be given when we translate from program graph to machine graph.

3.1 Basic Schemata

Translations from Id Nouveau to program graphs are most easily expressed pictorially. We will represent instructions as boxes, and arcs as lines:

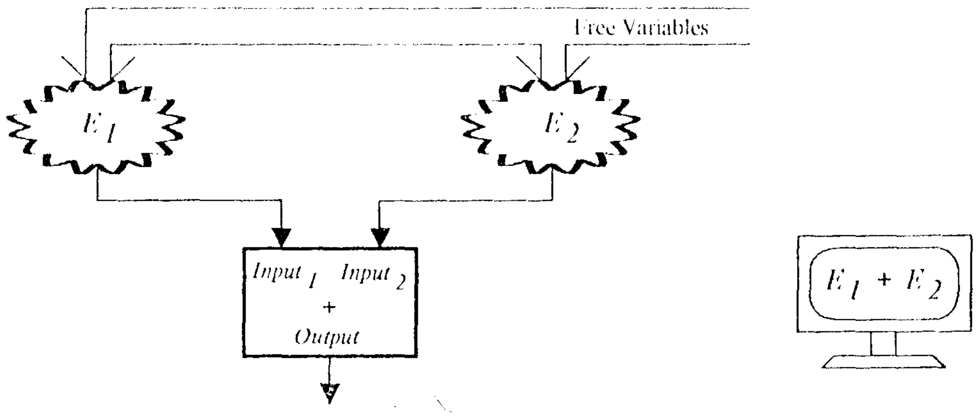


When we need to indicate an entire subgraph, we use a "blob":

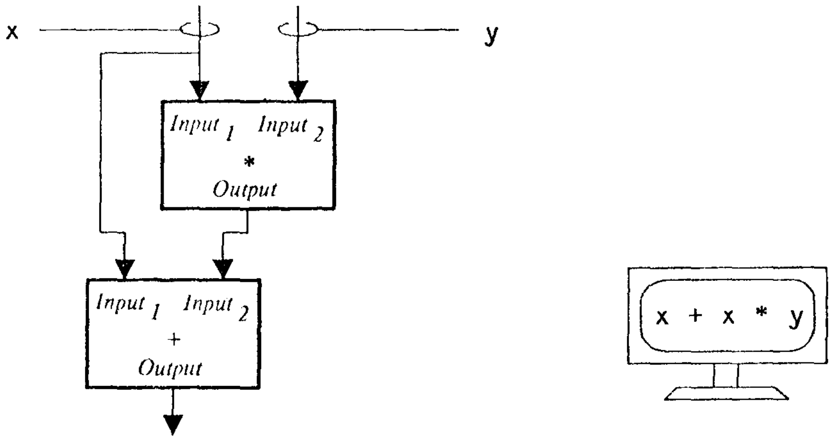


The double line indicates a collection of arcs — a bus, if you will. We are always careful to account for *all* arcs entering and leaving a subgraph, so that all interconnections are explicit.

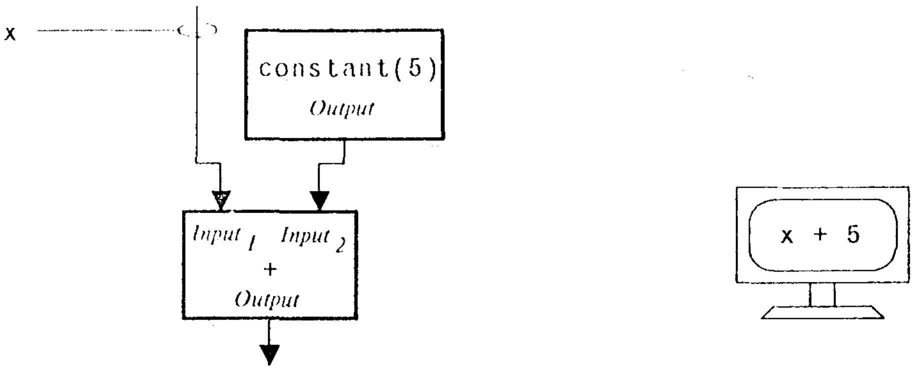
Expressions involving the arithmetic, relational, and logical operators (+, <=, and, *etc.*) are compiled in the straightforward way:



The arcs entering the subexpressions are precisely their free variables, as illustrated in the following example.



Constants in expressions are indicated by a special inputless constant instruction:



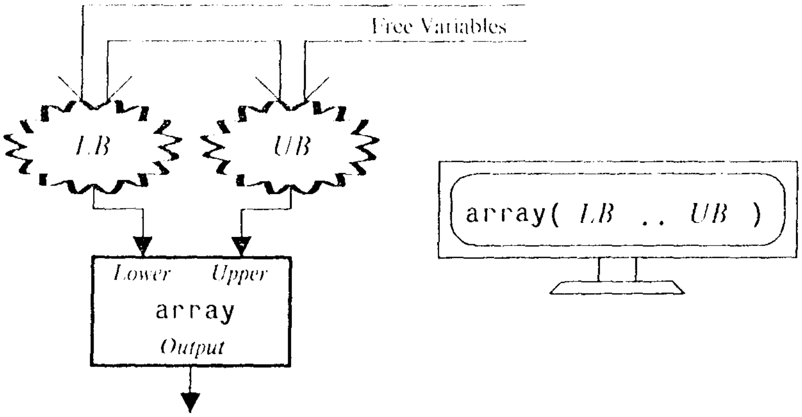
There are really an infinite number of `constant` opcodes, one for every possible constant. The firing rule for the `constant(5)` instruction is:

——— **Output *Output* of** `Constant(5)` ———

Pre-Condition: Output needed. **Value Produced:** 5

The precondition here is a bit strange — the `constant` instruction magically emits a token whenever one is needed by the instructions connected to it. As you might expect, we will need to be a little less vague about this when we convert to the machine graph.

We translate other constructs of Id Nouveau with the aid of a whole repertoire of program graph instructions. First, `array` expressions:



——— **Side-Effect of** `array` ———

Pre-Condition: *Lower* and *Upper* present.

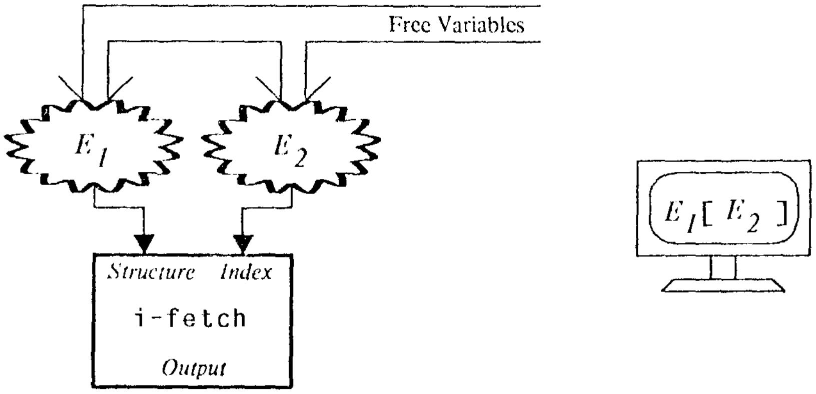
Effect: New empty I-structure is allocated from available I-structure memory.

——— **Output** *Output* **of** `array` ———

Pre-Condition: *Lower* and *Upper* present and new I-structure allocated.

Value Produced: (A descriptor for) the newly allocated I-structure.

I-structure fetch expressions:



The `i-fetch` instruction has an unusual firing rule, owing to the special nature of I-structures:

—— **Output** ***Output*** **of** I-Fetch ——

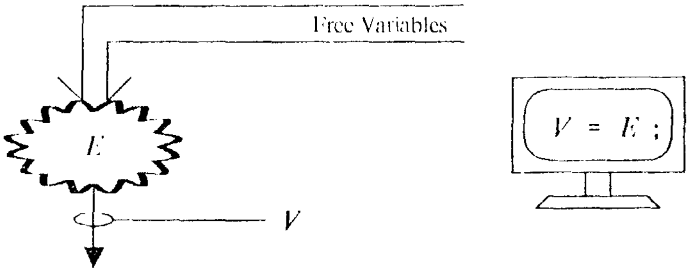
Pre-Condition: *Structure* and *Index* present, and element *Index* of *Structure* written.

Value Produced: Element *Index* of *Structure*.

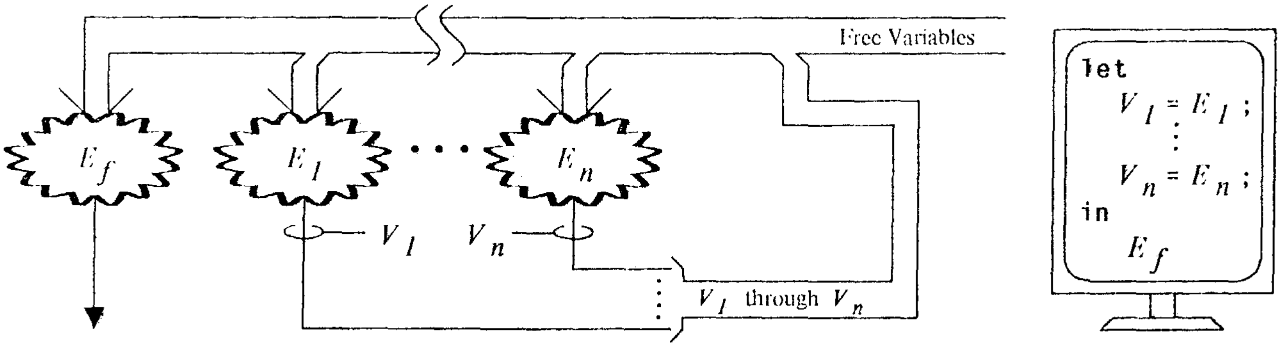
The construct for storing into I-structures is found in the next section.

3.2 *Let* Expressions

The simplest `let` expression introduces no new instructions of its own, but allows outputs of expressions to be named and used in more than one place. Each "ordinary" binding of a `let` is compiled as follows:

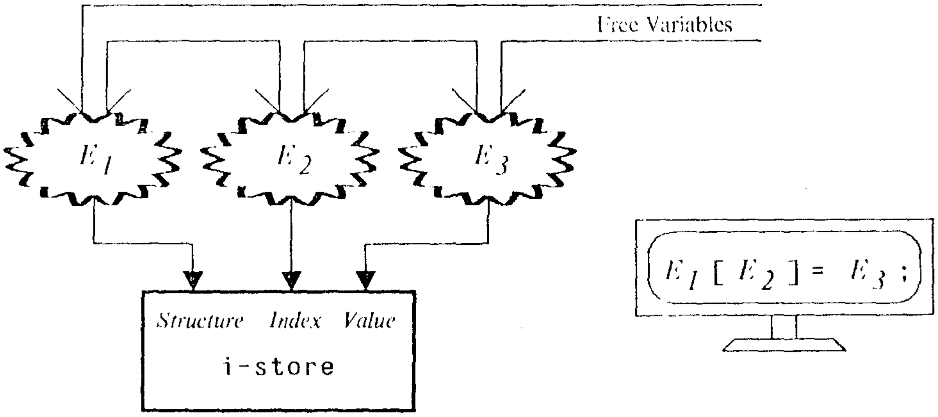


Notice that the output arc has been labeled with the variable name appearing on the left hand side. Labels on outputs are paired up with matching labels on inputs in the following schema for `let`:



The labeled arcs supply some of the free variables of the bindings' right hand sides as well as those of the `in` expression. This is consistent with the "letrec" scoping rule for `let` expressions. Any free variables not corresponding to the arcs labeled V_1 through V_n are exactly the free variables of the `let` expression as a whole.

Besides ordinary bindings, two other kinds of statements can appear in `let` expressions. These do not bind variables, but indicate side-effects to be brought about. The first of these causes an I-structure element to be written.



Yet another program graph instruction makes its debut; this one acts entirely by side effect:

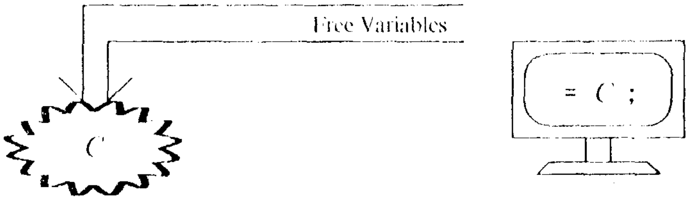
——— **Side-Effect of `i-store`** ———

Pre-Condition: *Structure*, *Index*, and *Value* present.

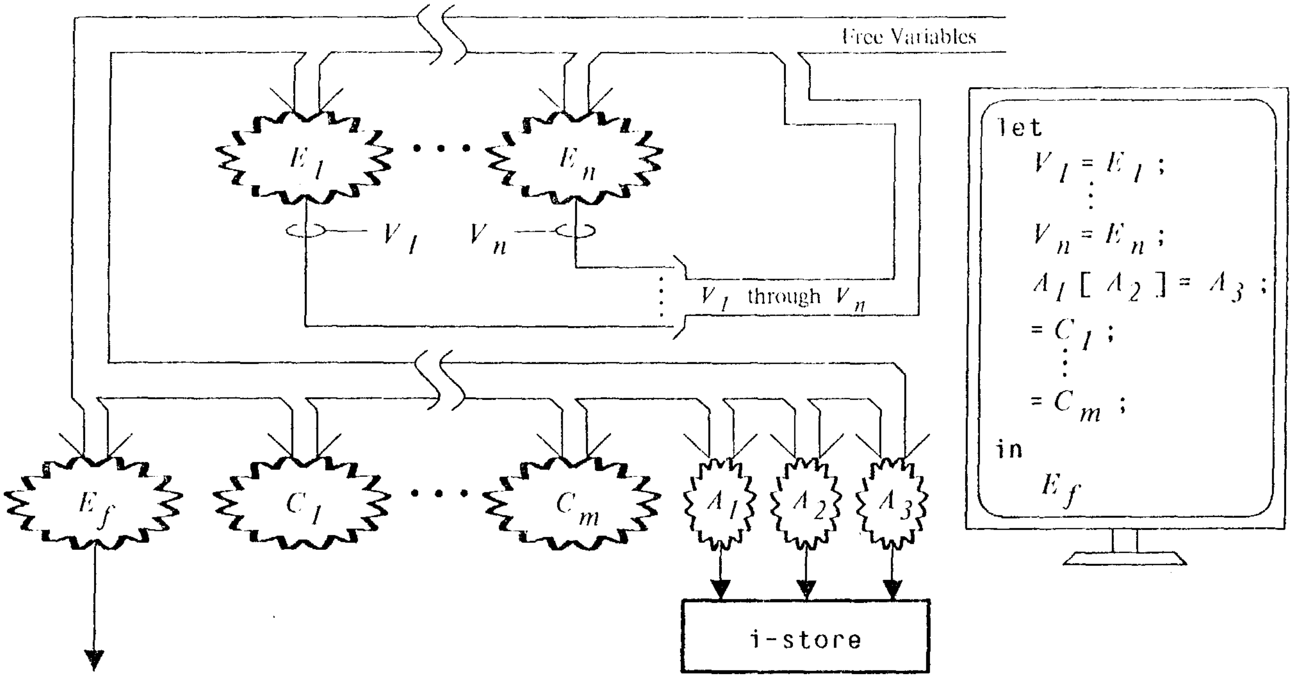
Effect: If element *Index* of *Structure* was not yet written, *Value* is written there, otherwise an error flag is raised.

The I-store statement is unusual in that variables on the left hand side contribute to the free variables of the statement. This is because the left hand side serves not to name the computations of the right hand side, as in the ordinary binding, but to indicate how to obtain a I-structure and its indices for writing. Since the I-store statement just indicates a side-effect, it has no outputs of its own.

The final kind of statement that can appear in a `let` is the "side-effect", or "command" statement. Here, we just compile the right hand side, which ought not to produce a value.



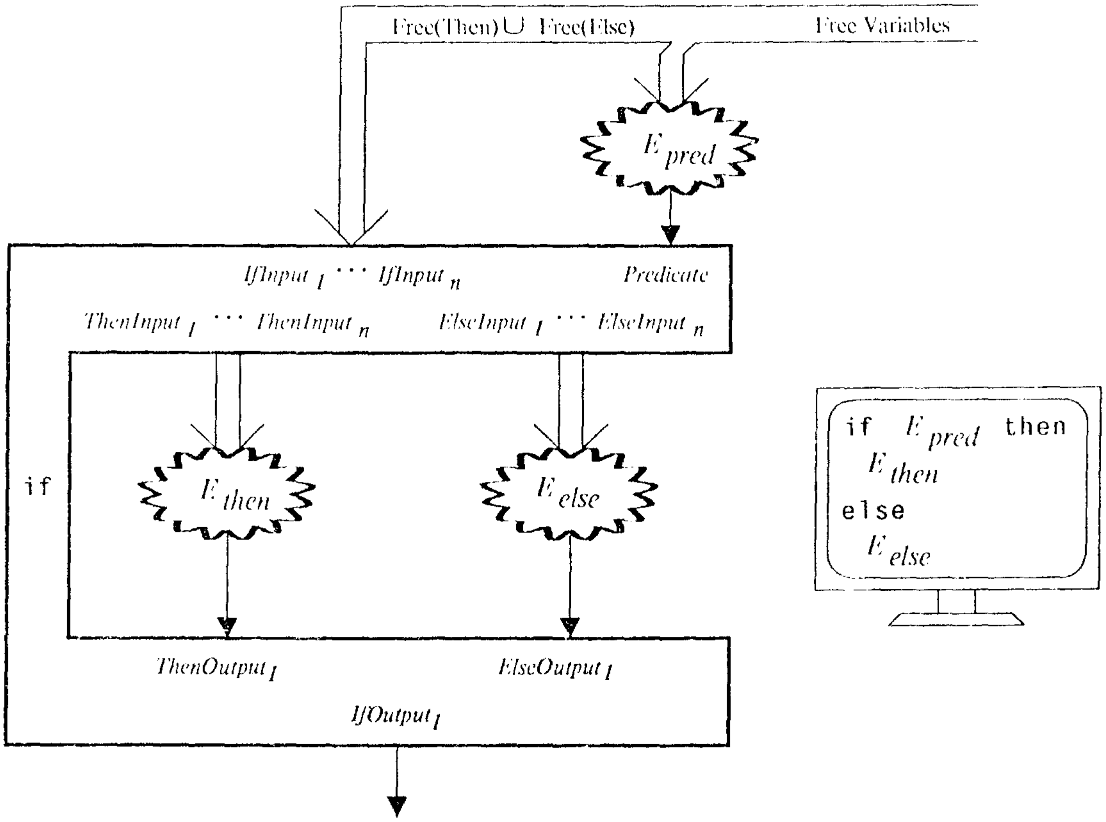
Putting it all together gives the final schema for `let` expressions/commands:



As a final note, we point out that the blob for the `in` expression will be absent if the symbol "`()`" follows the keyword `in`. In that case, the `let` is a command, and may appear on the right hand side of a command statement.

3.3 *If* Expressions

Now we tackle a more complicated program construct: the if expression. The schema for if is:



if E_{pred} then
  E_{then}
else
  E_{else}

All the work is done by a complicated program graph instruction called if. Basically, the idea is that the if instruction diverts tokens to either the "then" side or the "else" side depending on the token it receives on its *Predicate* input. It also merges the results from the two sides so that the value computed by the if expression flows along the single output arc. All of this is summarized by the firing rules for if below:

——— **Output** $ThenInput_i$ **of** if ———

Pre-Condition: $IfInput_i$ present and true present on *Predicate.*

Value Produced: $IfInput_i$

——— **Output** ***ElseInput**$_i$* **of** `if` ———

Pre-Condition: *IfInput$_i$* present and `false` present on *Predicate*.

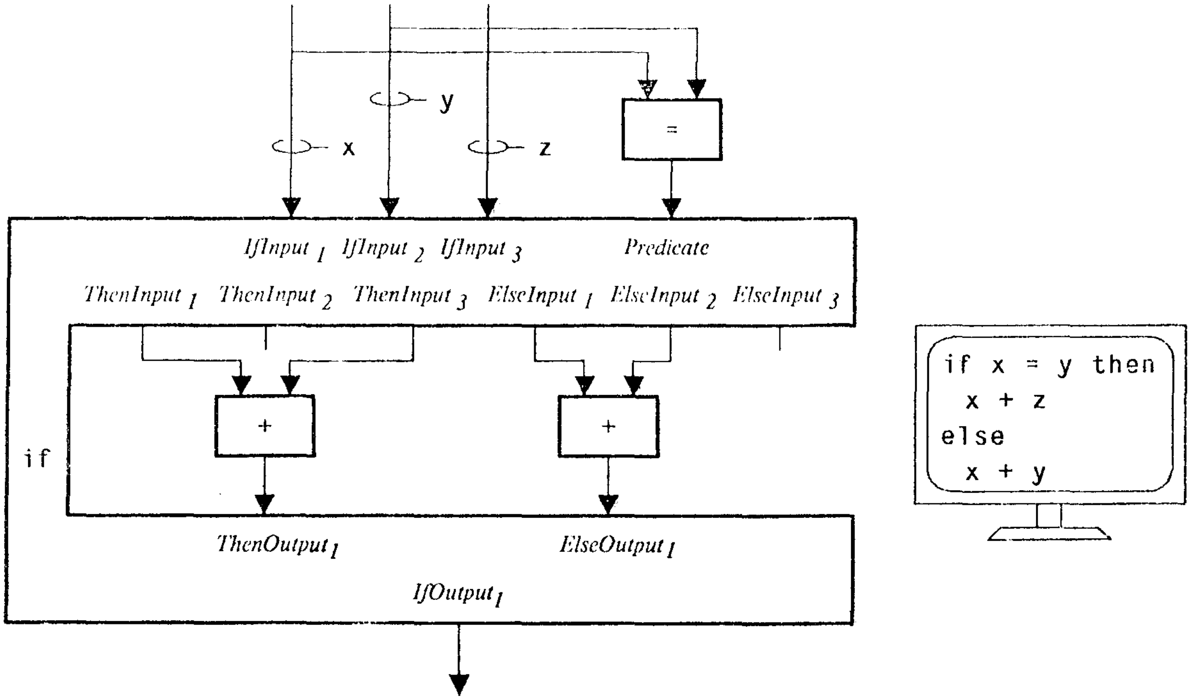
Value Produced: *IfInput$_i$*

——— **Output** ***IfOutput**$_i$* **of** `if` ———

Pre-Condition: *ThenOutput$_i$* present when `true` present on *Predicate* or *ElseOutput$_i$* present when `false` present on *Predicate.*

Value Produced: *ThenOutput$_i$* (*Predicate* `true`) or *ElseOutput$_i$* (*Predicate* `false`)

The inputs to the `if` are all variables that might be needed by either the "then" side or the "else" side — in other words, the union of the free variables of the two sides. Of course, not all of these variables need be connected to both sides within the `if`. Here is an example of an `if` expression:

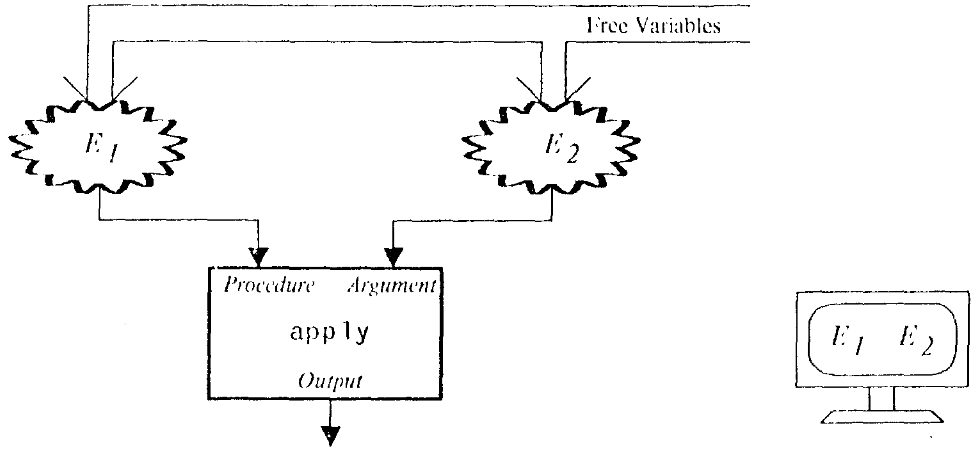


```
if x = y then
  x + z
else
  x + y
```

One thing the firing rule for `if` does *not* require is that all inputs arrive at the `if` before being routed to the appropriate side. For example, if x were 3 and y were 4 in the example above, a token carrying 7 can appear at the output even if the token for z had not yet arrived.

3.4 Procedures and Applications

Applications are simple in appearance, remembering that due to currying an expression like `( f a b c )` appears as three left-associated applications.



The firing rule for `apply`:

——— **Output** *Output* **of** `apply` ———

Pre-Condition: *Procedure* present

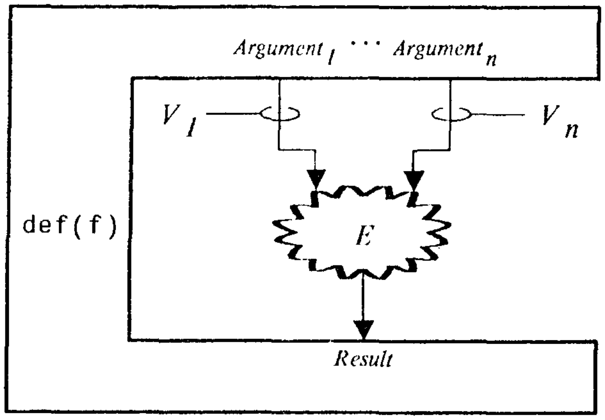
Value Produced: Result of applying *Procedure* to *Argument*

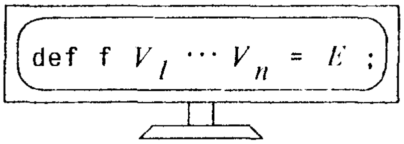
The firing rule for `apply` is deceptively simple. If *Procedure* is a procedure of one argument, then `apply` is fairly straightforward: it simply passes the argument to the procedure, and sends the result along its *Output* arc. If *Procedure* is a procedure of two arguments, say, then we are only applying the first argument here. In keeping with the curried definition of procedures, then, we must return an object which when applied to another argument invokes *Procedure* with both arguments. In short, we return a *closure* of *Procedure* over its first argument. The *Procedure* input of `apply`, therefore, expects a closure.

Without giving away too many details of the implementation, we can say that a closure has at least three components. First, it carries the name of the procedure ultimately to be invoked. Second, it indicates how many arguments remain before the procedure's arity is satisfied. Third, it holds all arguments accumulated thus far; their number plus the number remaining always adds up to the arity of the procedure. The action taken by `apply` depends on the number of arguments not yet collected. If one, then it invokes the procedure, sending *Argument* and any arguments recorded in the closure to the procedure. If two or more, it creates a new closure containing the new argument as well as any in the old closure, whose "number of arguments remaining" field is one less. Whenever a program refers to the name of a procedure, it is really referring to a closure for that procedure with no arguments collected.

We want procedures to be *non-strict* in that they can begin to execute even though some of their arguments have yet to be computed. This is expressed in the firing rule by the absence of a precondition requiring *Argument* to be present before producing an output. This means that as soon as we have the *Procedure* argument and see that its arity is not yet satisfied, we create a new closure which will eventually contain the new argument whenever it arrives. Or, if its arity is satisfied, we invoke the function, and whenever the argument arrives it will automatically be diverted to the procedure. Exactly how we accomplish this feat will be explained when we turn to the machine graph (hint: it involves the use of I-structures).

All that remains is to describe how we represent procedure definitions themselves in the program graph. Not surprisingly, we use a `def` instruction:





`def f` $V_1 \cdots V_n$ `=` E `;`

`Def` is a bit tricky to explain by firing rule. Let it suffice to say that sometime after the procedure is invoked an argument pops out of each of the $Argument_i$ outputs, and when the body sends a result to the $Result_i$ input it is routed back to the `apply` instruction that triggered the invocation.

3.5 Loops

The schemata given in the preceding sections are sufficient to implement all of Id Kernel, and therefore all of Id Nouveau, assuming we transform `for` and `while` loops into recursive procedures. In practice, however, there are two reasons for giving loops special treatment. First, the tagged-token dataflow architecture, like most dataflow architectures, provides a particularly efficient mechanism for executing loops. Second, the program graph representation of loops admits the possibility of several optimizing transformations, such as invariant code motion.

In this section, therefore, we give a program graph schema for `while` loops (we assume that `for` loops are converted to their `while` loop equivalents). This does not imply, however, that programmers who prefer expressing their programs as recursion will obtain lower performance than those who prefer the `while` and `for` constructs. In many cases it will be possible for a good compiler to convert recursive programs into loops [Arsac 82]. Conversely, we may choose to retain loops in the program graph for purposes of optimization, and then convert them to recursion before translating into machine code. Unfortunately, we will limit our flexibility somewhat by assigning slightly different semantics to the program graph's loop construct as compared to the corresponding recursion; this is discussed in detail below.

The program graph representation of loops is in the same spirit as that for conditionals: we introduce a new instruction called `loop` which isolates the different components of the loop. Recall that a `while` loop has three parts: a *predicate* that determines when the loop terminates, a *body* that is evaluated each time the predicate is true, and a `return` expression that is evaluated after the predicate turns false. The body of a loop is a statement list that is similar to the statement list in a `let` expression. In addition to all the statement types discussed for `let`, however, a loop statement list can also contain bindings of the form `new` *Variable* = *Expression*; we call these bindings "`new` bindings".

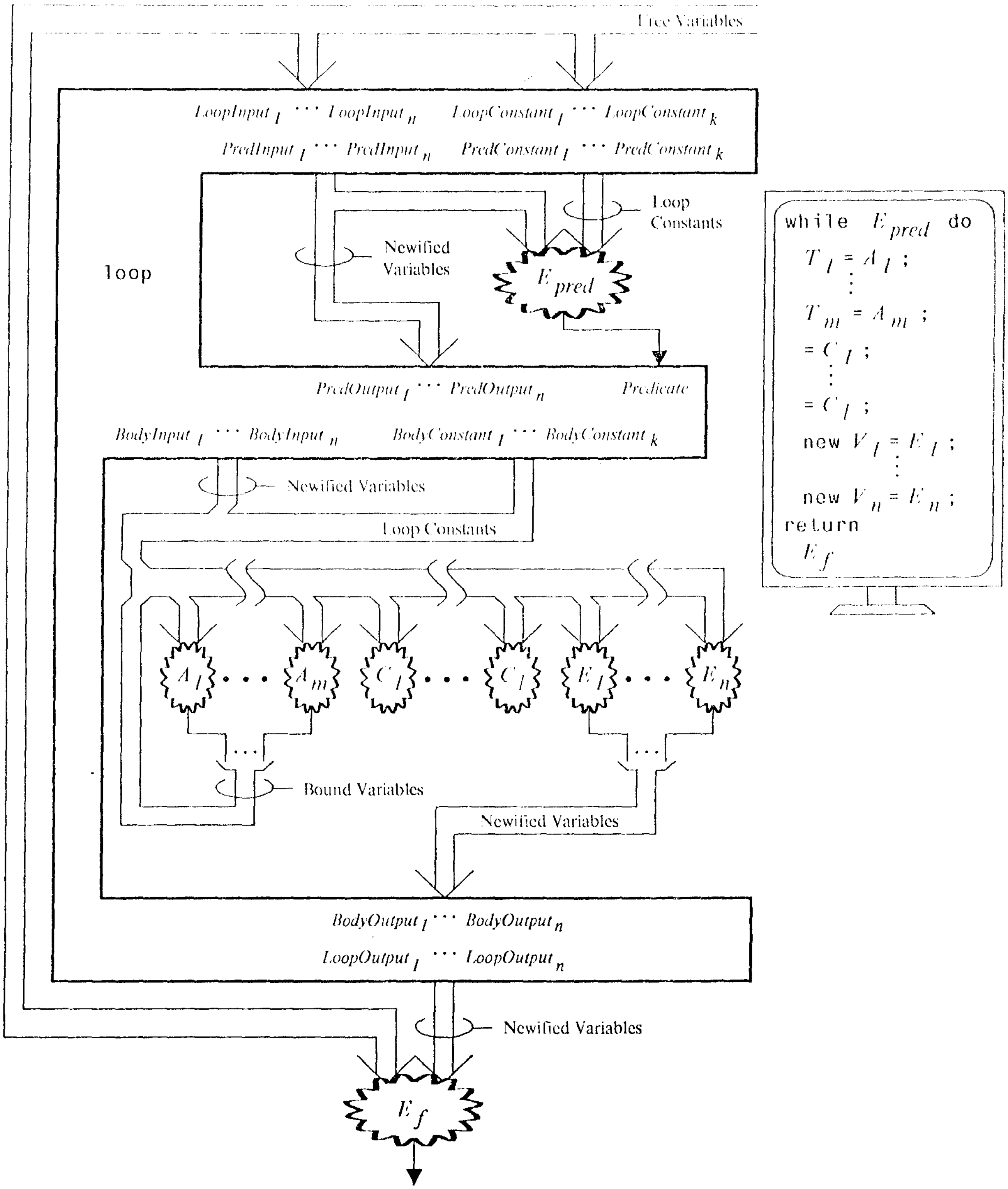
The variables used in the predicate and body of a loop can be classified into three categories:

Newified variables Variables occurring on the left hand sides of `new` bindings in the body. These variables pass information from one iteration of the loop to the next, and from the final iteration to the `return` expression.

Bound variables Variables occurring on the left hand sides of ordinary bindings in the body. These variables hold temporary results *within* a given iteration. Unlike newified variables, they cannot be sensibly referenced from the predicate or return expression.

Loop constants Any free variables of the predicate or body not falling into one of the two previous categories. They are called loop constants because their value cannot change from one iteration to the next.

The schema for while loops uses an elaborate program graph instruction called loop.


Free Variables
$LoopInput_1$ ⋯ $LoopInput_n$
$LoopConstant_1$ ⋯ $LoopConstant_k$
$PredInput_1$ ⋯ $PredInput_n$
$PredConstant_1$ ⋯ $PredConstant_k$
Loop Constants
Newified Variables
loop
E_{pred}
$PredOutput_1$ ⋯ $PredOutput_n$
Predicate
$BodyInput_1$ ⋯ $BodyInput_n$
$BodyConstant_1$ ⋯ $BodyConstant_k$
Newified Variables
Loop Constants
A_1 • • • A_m
C_1 • • • C_l
E_1 • • • E_n
Bound Variables
Newified Variables
$BodyOutput_1$ ⋯ $BodyOutput_n$
$LoopOutput_1$ ⋯ $LoopOutput_n$
Newified Variables
E_f

```
while $E_{pred}$ do
  $T_1 = A_1$ ;
      :
  $T_m = A_m$ ;
  $= C_1$ ;
      :
  $= C_l$ ;
  new $V_1 = E_1$ ;
      :
  new $V_n = E_n$ ;
return
  $E_f$
```

(Although omitted from the figure for reasons of space. I structure Store statements are of course permitted in the loop's body.) For each newified variable, there are corresponding *LoopInput*, *PredicateInput*, *PredicateOutput*, *BodyInput*, *BodyOutput*, and *LoopOutput* ports, through which the variable is shuttled through the various components of the loop. Bound variables are treated just as they are in `let` expressions: they are fed back to all statements in the body. Loop constants are wired to special *LoopConstant* inputs; more on that later.

Tokens entering the *LoopInput* inputs of a `loop` are first routed to the predicate, which decides if the loop body is to be executed. If not, the tokens are diverted to the *LoopOutput* outputs of the `loop` where they are consumed by the `return` expression. If, on the other hand, the predicate is `true`, the tokens are sent into the loop body, where they are used as the values of the newified variables. During execution of the body, values for bound variables are computed and used in other body computations. The body may perform some side effects, but ultimately it computes new values for the newified variables; these are sent to the *BodyOutput* inputs of the loop. From there, they are once again fed to the predicate, and the process repeats until the predicate evaluates `false`. When tokens finally leave the *LoopOutput* outputs, the predicate will have been executed at least once, and the body exactly one fewer time than the predicate.

Loop constants arrive on the *LoopConstant* inputs of the `loop`. They are emitted at the *PredicateConstant* and *BodyConstant* outputs each time the predicate (body) executes. While we could have circulated the loop constants the same way we did the newified variables, giving them special status allows us to exploit their constantness when translating to machine code and during program optimization.

With this informal description of `loop` in mind, here are the firing rules for `loop`:

——— **Output** *PredicateInput*$_i$ **of** `loop` ———

Pre-Condition: All *LoopConstant* present and either all *LoopInput* present (first iteration) or all *BodyOutput* present (succeeding iterations).

Value Produced: *LoopInput*$_i$ or *BodyOutput*$_i$, as appropriate.

——— **Output** *PredicateConstant*$_i$ **of** `loop` ———

Pre-Condition: All *LoopConstant* present and either all *LoopInput* present (first iteration) or all *BodyOutput* present (succeeding iterations).

Value Produced: *LoopConstant*$_i$

——— **Output** *BodyInput*$_i$ **of** `loop` ———

Pre-Condition: All *PredicateOutput* present and `true` present on *Predicate.*

Value Produced: *PredicateOutput*$_i$

——— **Output** *BodyConstant*$_i$ **of** `loop` ———

Pre-Condition: All *PredicateOutput* present and `true` present on *Predicate.*

Value Produced: *LoopConstant*$_i$

——— **Output** *LoopOutput*$_i$ **of** `loop` ———

Pre-Condition: All *PredicateOutput* present and `false` present on *Predicate.*

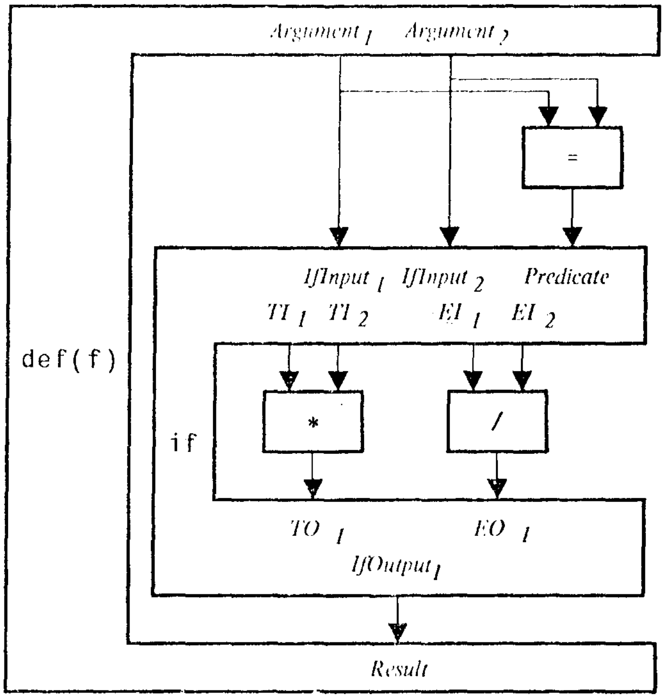
Value Produced: *PredicateOutput*$_i$

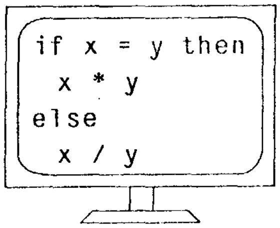
The astute reader will notice that according to the firing rules above, an iteration cannot begin until the previous iteration is completed, nor can any iteration proceed until all loop constants have arrived. This is the semantic difference between loops and recursion that was alluded to earlier. While this distinction is unsatisfactory from a purist standpoint, it is crucial to the tagged-token dataflow architecture's ability to execute loops efficiently. Current experience shows that the restriction is fairly inconsequential, as it only tends to rule out programs that most people would consider bizarre anyway, such as the following:

```
def backwards_fill (n) =
  let
    a = array(1..n);
    a[n] = n;
    sum = 0;
    i = 1
  in
    while i < n do
      val = a[i+1];
      a[i] = val;
      new sum = sum + val;
      new i = i + 1
    return a, sum
```

In any iteration, the value to be computed for `val` is data dependent on the value computed in the *succeeding* iteration (except in the last iteration, where it depends only on the value of `n`), and so this program will not execute if we require one iteration to be processed at a time.

From the foregoing, it might appear that the dataflow implementation of loops will execute loops in a sequential fashion. On the contrary, in Section 7.4 we will show how several iterations can be executed in parallel. The key point is that we will only allow a *fixed* number of iterations to be outstanding at any given time, whereas programs like backwards_fill require an *arbitrary* number of concurrent iterations. Furthermore, the decision as to how many concurrent iterations we allow will be deferred until run time, and their number might be as small as one. If program graphs are to be useful in proving correctness of compiler optimizations, we must adopt the most stringent firing rules for loop. Any program that terminates with no concurrent iterations is guaranteed to terminate and produce the same answer if more than one concurrent iteration is allowed.

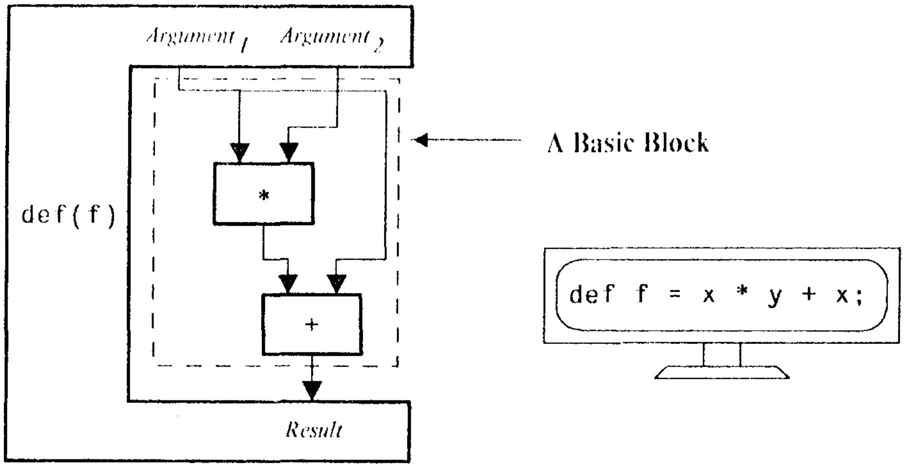




```
if x = y then
  x * y
else
  x / y
```

This program has three basic blocks: the then side of the if, the else side of the if, and the entire procedure body itself. Consider now the behavior of the procedure body basic block. If we drop a token into each of its two inputs we always get a token at the output. On the other hand, it is not strictly true that all instructions execute, since either the * or the / will not fire, depending on whether x equals y. Hence, the procedure body doesn't quite meet our definition of a basic block. A correct definition of basic blocks must treat encapsulators in a special way.

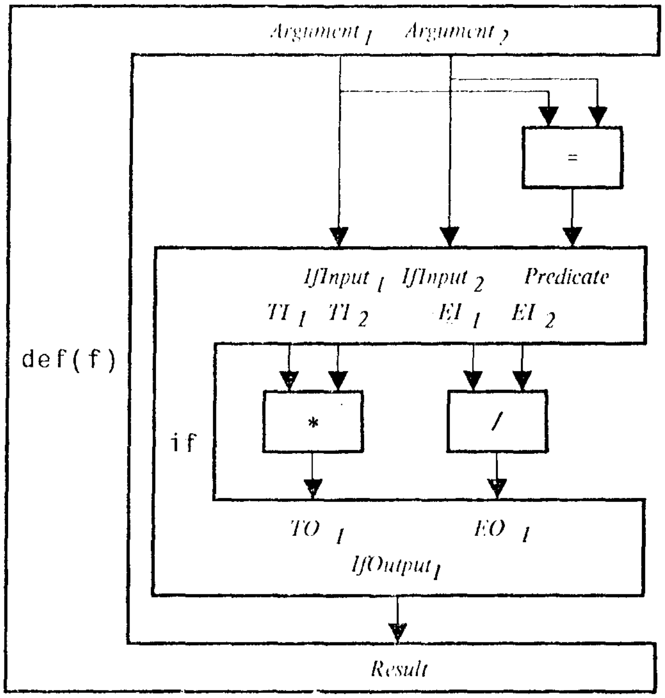
The simplest basic block is an individual non-encapsulator instruction such as + or i-store, since these possess the one-in-one-out property and (trivially) the complete unique execution property. Furthermore, it is easy to see that any composition of basic blocks is also a basic block, as long as no cyclic dependencies are introduced. Finally, we observe that encapsulators have the one-in-one-out property; in the previous example the if will always produce an output when it receives a token on each of its three inputs. We can also attribute the complete unique execution property to an encapsulator *as long as we ignore its interior.* In other words, when viewing encapsulators from the outside we will treat them as if they were single instructions.

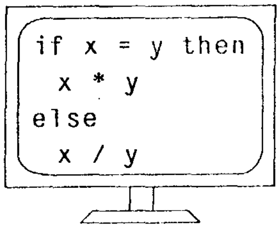


```
def f = x * y + x;
```

It is easy to see that the entire body is indeed a basic block. In fact, basic blocks arise in five places within program graphs: the bodies of procedures, the then and else subgraphs of ifs, and the predicate and body subgraphs of loops. Given that, it is clear that the def, if, and loop program graph instructions encapsulate basic blocks, and so we will collectively call them *encapsulators.*

Here is a more complicated program:





```
if x = y then
  x * y
else
  x / y
```

This program has three basic blocks: the then side of the if, the else side of the if, and the entire procedure body itself. Consider now the behavior of the procedure body basic block. If we drop a token into each of its two inputs we always get a token at the output. On the other hand, it is not strictly true that all instructions execute, since either the * or the / will not fire, depending on whether x equals y. Hence, the procedure body doesn't quite meet our definition of a basic block. A correct definition of basic blocks must treat encapsulators in a special way.

The simplest basic block is an individual non-encapsulator instruction such as + or i-store, since these possess the one-in-one-out property and (trivially) the complete unique execution property. Furthermore, it is easy to see that any composition of basic blocks is also a basic block, as long as no cyclic dependencies are introduced. Finally, we observe that encapsulators have the one-in-one-out property; in the previous example the if will always produce an output when it receives a token on each of its three inputs. We can also attribute the complete unique execution property to an encapsulator *as long as we ignore its interior.* In other words, when viewing encapsulators from the outside we will treat them as if they were single instructions.

Given the foregoing, we amend our definition of basic blocks as follows: a basic block is a subgraph such that if one token is fed to each input of the block, every instruction in the block will eventually execute exactly once, and one token will appear at each output of the block, where encapsulators within the block are viewed as "black box" instructions, from their exterior only.

Encapsulators in program graphs play the same role as do control flow arcs in a conventional compiler's flow graphs: they regulate the initiation of basic blocks. Their power lies in their ability to be treated in the same manner as individual instructions when only the surrounding region of code is of interest; this makes it possible, for example, to move entire conditionals or loops in the same way that code motion is accomplished for ordinary instructions. This capability was recognized in the work of Ottenstein [Ferrante 83], whose "extended dataflow graphs" reflect the encapsulator idea. In Ottenstein's scheme, however, encapsulators are not manifest but only implied by control flow arcs which augment the data flow arcs. Here, encapsulators encode control flow *as* data flow, leading to a more consistent treatment of blocks.

4.2 Optimizations Within Basic Blocks

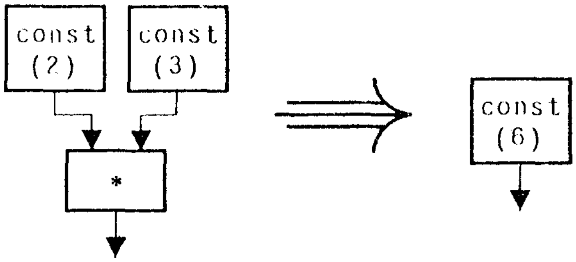
Basic blocks in program graphs are nearly identical to the directed acyclic graph (dag) representation of basic blocks used by conventional compilers [Aho 86]. Consequently, the same optimization techniques a conventional compiler applies to dags can be applied to basic blocks within a program graph. We are at an advantage, however, since any transformation of the program graph is a transformation of the program itself. In a conventional compiler the dag is usually an auxiliary data structure, which after optimization must be converted back to the compiler's intermediate program form (*e.g.*, quadruples). We also benefit from the use of encapsulators, since they allow us to treat whole regions of code as single instructions. Often we never need to distinguish between instructions and encapsulators, save that broad characteristics of an encapsulator will sometimes be determined by the characteristics of its interior. The lack of assignment and unrestricted control flow (indeed, of *any* control flow) in Id Nouveau contributes to the simplicity of the optimizations presented here compared to their counterparts in conventional compilers. I-structures make the problems somewhat more interesting than for a purely functional language.

The applicability of many optimizations depends on whether the instructions involved cause side-effects. The meaning of "side-effects" is very clear in the context of a program graph: an instruction causes a side-effect if and only if its execution can be detected by another instruction even though no explicit data flow arc exists between them. There are only three program graph instructions which can cause side-effects: `i-store`, `array`, and `apply`. The side effect of `i-store` is quite clear: it writes an I-structure location, which can affect the operation of other `i-fetch` instructions in the graph. The side effect of `array` is a bit subtler; when an `array` instruction executes, returning an empty I-structure, it also affects other `array` instructions by preventing them from receiving the same I-structure. If this seems a bit slippery, consider that two `array` instructions, both receiving the identical arguments `1` and `10`, say, will return *different* I-structures. Clearly this could not be possible if `array` were purely functional (side effect free). Finally, `apply` causes side-effects when it invokes a procedure whose body has side effects. From the caller's point of view, any side effects brought about by the invoked procedure are due to the `apply` which caused the invocation. Interprocedural analysis is very useful in determining which `apply`s cause side effects and which don't; such analysis is explored in Section 4.4.

In addition to `i-store`, `array`, and `apply`, when we view an encapsulator as a single instruction we must consider it to have side effects if it encapsulates any instructions which cause side effects, at least in the worst case.

4.2.1 Constant Folding

A very simple optimization is *constant folding* [Aho 86, Allen 72], in which expressions involving only constants are evaluated at compile time. This is trivial in a dataflow compiler: the basic block is simply searched for instructions all of whose inputs come from `constant` instructions. An example:

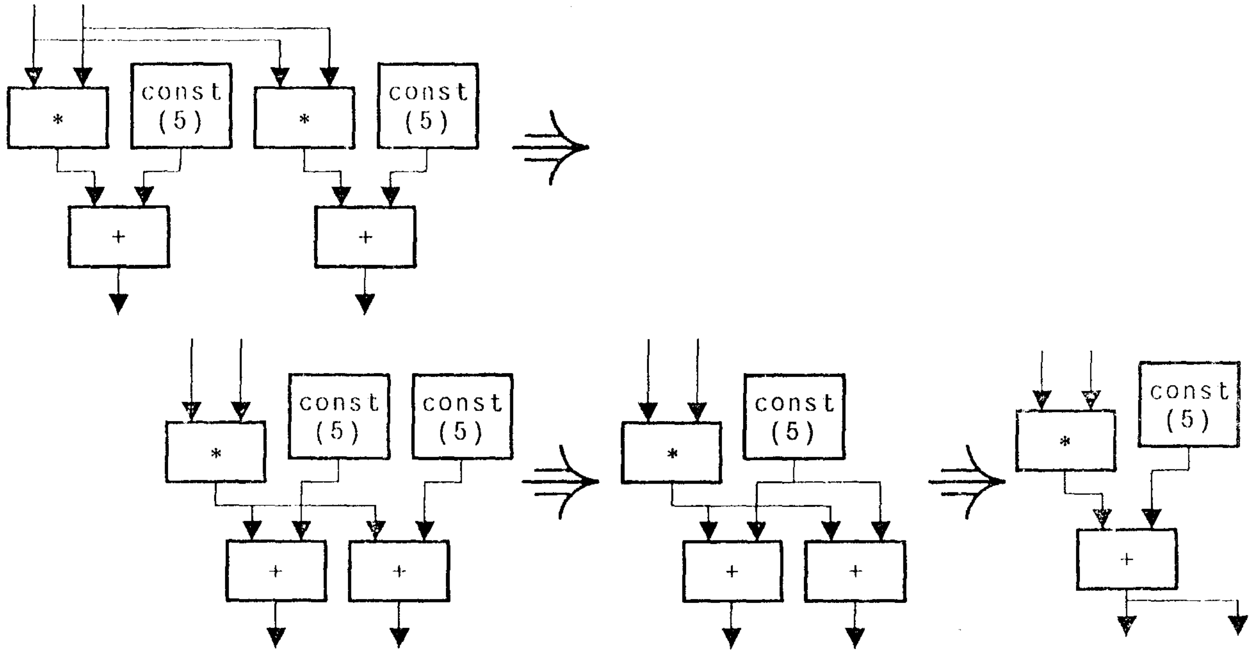


Of course, only side-effect free instructions are candidates for folding. An instruction with side effects such as `array` cannot be folded, since it may execute several times (if it were in a loop, say), returning a different value each time even though its arguments remain unchanged.

Constant folding is typically augmented with an assortment of algebraic transformations, such as replacing `x * 1` with `x` or taking advantage of associativity and commutivity. All of these are equally applicable to program graphs.

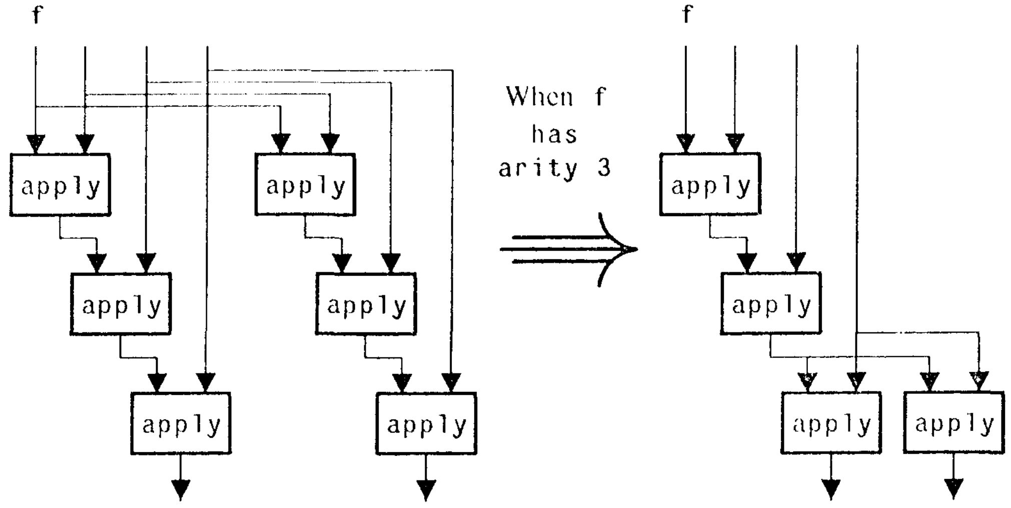
4.2.2 Common Subexpression Elimination

A second intra-block optimization is *common subexpression elimination*, which tries to avoid repeated computation of the same value. Again, this is quite simple in the program graph representation: we search for groups of instructions bearing the same opcode and whose inputs come from the same place, and for equivalent `constant` instructions. The search may be performed efficiently by hashing on a key derived from the opcode and inputs of each instruction. Common subexpressions can propagate downward, as illustrated below.



Again, our optimization is limited to those instructions that do not cause side-effects; we cannot collapse two `array` instructions, for example, as they must return different I-structures even though their bounds are the same.

Lacking any evidence to the contrary, we must assume that `apply` instructions cause side effects, and are therefore not candidates for common subexpression elimination. As mentioned earlier, interprocedural analysis can sometimes supply the necessary information. It should be noted, however, that even if a procedure `f` has side effects, only the last `apply` in the chain that collects its arguments need be treated as an instruction with side-effects. This is because all `apply`s save the last simply create a new closure given an old closure and a new argument, which is a completely functional operation. The final `apply` actually invokes the procedure, and so it alone appears to cause the side effects. An example:



This optimization, of course, can only be attempted if it can be ascertained which `apply` actually causes invocation; *i.e.*, if the arity of `f` is known.[3]

Considering encapsulators as single instructions, it is possible to combine two `if` or `loop` instructions as common subexpressions if all of their exterior inputs come from the same sources. Of course, there is the additional proviso that the interiors be identical, and free of side-effects. To be truly effective, we must also be prepared to consider encapsulators that are the same but for permutations of their inputs and outputs.

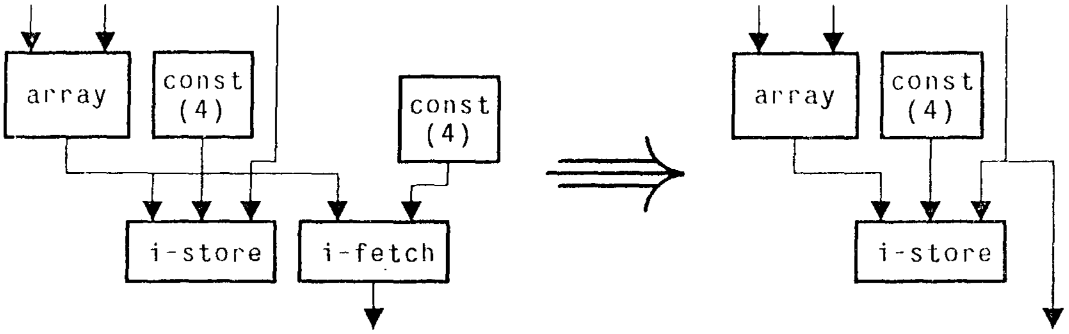
The common subexpression elimination algorithm was also reported in [Skedzielewski 85a], although in that work side effects were not considered.

4.2.3 I-Fetch Elimination

I-fetch elimination attempts to bypass fetching from an I-structure when it can be determined that the data is already present on another arc within the block. If there is an `i-fetch` instruction and an `i-store` instruction whose *Structure* and *Index* inputs are fed from the same place and/or equivalent `constant` instructions, then the `i-fetch` is eliminated

[3] On the other hand, if the arity is known it is not clear that we would even want to compile the call as a chain of `apply`s in the first place; see Section 7.6.

and whatever was connected to its output is connected instead to whatever feeds the *Value* input of the i-store. This is illustrated below.



Normally, we must retain the i-store instructions since the structure may be used elsewhere, *e.g.* passed outside the basic block. On the other hand, the i-stores may turn out to be dead code, as explained in the next section. The combination of I-fetch Elimination and Dead Code Elimination can yield efficient code for program fragments involving tuples, as in the following (somewhat trivial) expression, where all I-structures could be removed.

```
let
  a, b = let
           x = q * 5
         in
           x, x * x;
  ...
```

Further opportunities for such optimizations can be exposed by code motion across ifs, as described in Section 4.3.1.

4.2.4 Dead Code Elimination

Any program graph instruction all of whose outputs are unconnected can be considered dead code, since no part of the program depends on its results. As always, the exceptions are instructions which cause side-effects; though its output be unconnected, an instruction causing a side-effect contributes to the program's computation. An unconnected array instruction, however, can be eliminated as dead code, since its effects are not felt unless its output is used. That is, an array instruction causes the side effect of allocating a region of I-structure memory, which is felt by other array instructions in that they can no longer obtain that particular region. On the other hand, a program cannot distinguish between regions of I-structure memory save that it is possible to tell whether two I-structures are the same region or not. Since execution of an array instruction affects only *which* regions other array instructions will receive, an unconnected array can be eliminated. This special treatment of allocation with respect to dead code elimination is also discussed in Steele [Steele 78].

An encapsulator with all exterior outputs unconnected can also be eliminated, provided that the only side-effect causing instructions in its interior are array instructions.

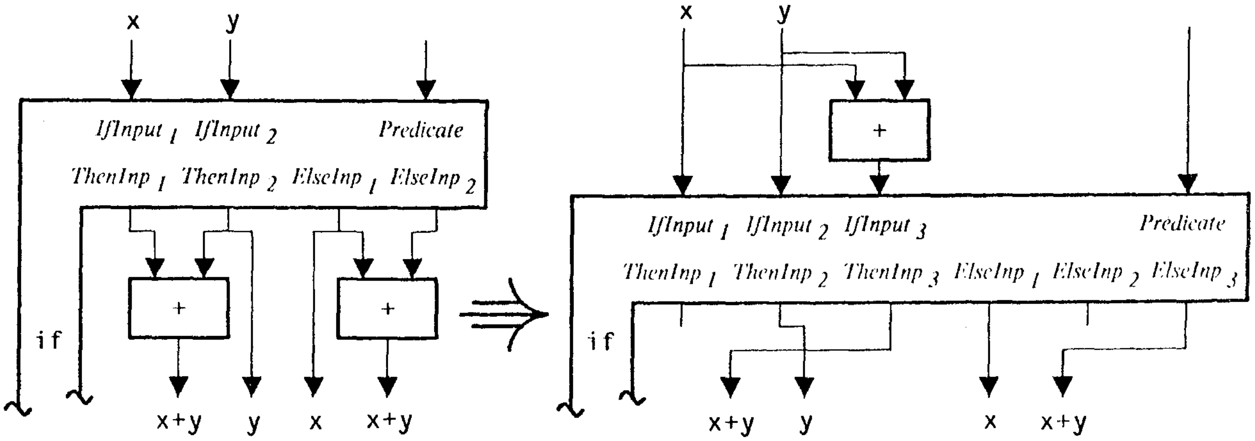
Besides unconnected instructions and encapsulators, there is another situation which can be considered dead code. If an array instruction is connected only to i-store instructions, then the array and all the i-stores can be eliminated, since the structure is useless if not read. This situation arises frequently if the I-fetch Elimination optimization described above is applied.

4.3 Optimizations Across Encapsulators

The optimizations discussed in the preceding section were all applied within a basic block. We can also perform transformations which move code between basic blocks, across the encapsulators which separate them. These kinds of transformations fall under the general category of *code motion.* Code motion is used to remove invariants from loops, and to bring pieces of code into the same block so that they may be subject to intra-block optimizations. We will concentrate on what kinds of code motion are possible and under what conditions it is safe; strategies for determining when code motion is desirable are beyond the scope of this thesis. We note in passing that the code motion algorithms of Ferrante and Ottenstein [Ferrante 83] should be directly applicable, since their representation is so similar to program graphs. Of course, the single-assignment nature of Id Nouveau eliminates the need for live/dead variable analysis and the other complications faced by their method and by compilers for imperative languages in general.

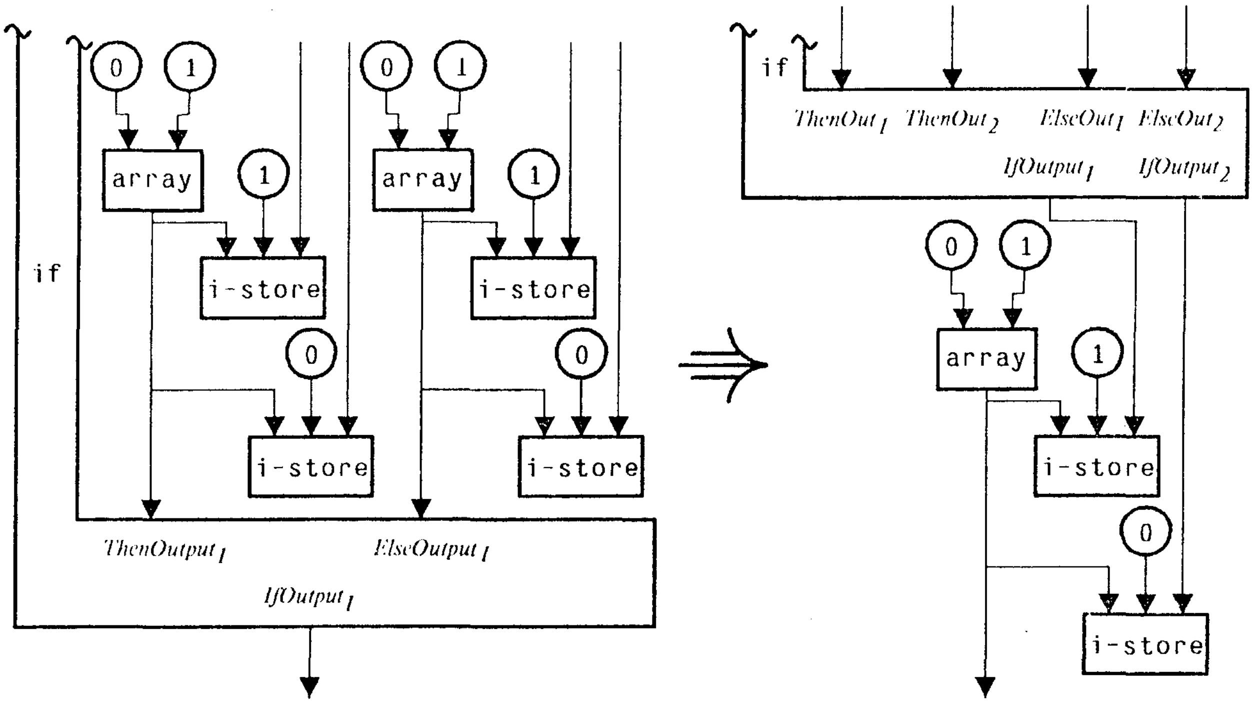
4.3.1 Code Motion Across *ifs*

If a subexpression appears in both arms of a conditional, and it depends only on variables computed outside the conditional, then the subexpression can be lifted out. In the following example, the expression `x + y` is lifted from both sides of an `if`:



The benefits of this transformation include the usual ones: the size of the code is reduced, and bringing the subexpression into the enclosing block may trigger further optimizations within that block. An additional benefit may accrue if we reduce the number of arcs which cross the `if`, for as we will see in a later chapter there is a certain amount of overhead for each of these arcs. In the example above, we eliminate one such arc if the only use of `x` and `y` within the `if` were in the lifted expressions. On the other hand, the inverse of this transformation — pushing instructions inside both arms of an `if` — can also reduce the number of arcs crossing the `if`. Evaluating the trade-offs can be quite difficult.

Code motion is also possible at the opposite boundary of `if`. Instructions can be pushed out the bottom of an `if`, for example, by adding additional outputs to the `if` encapsulator:



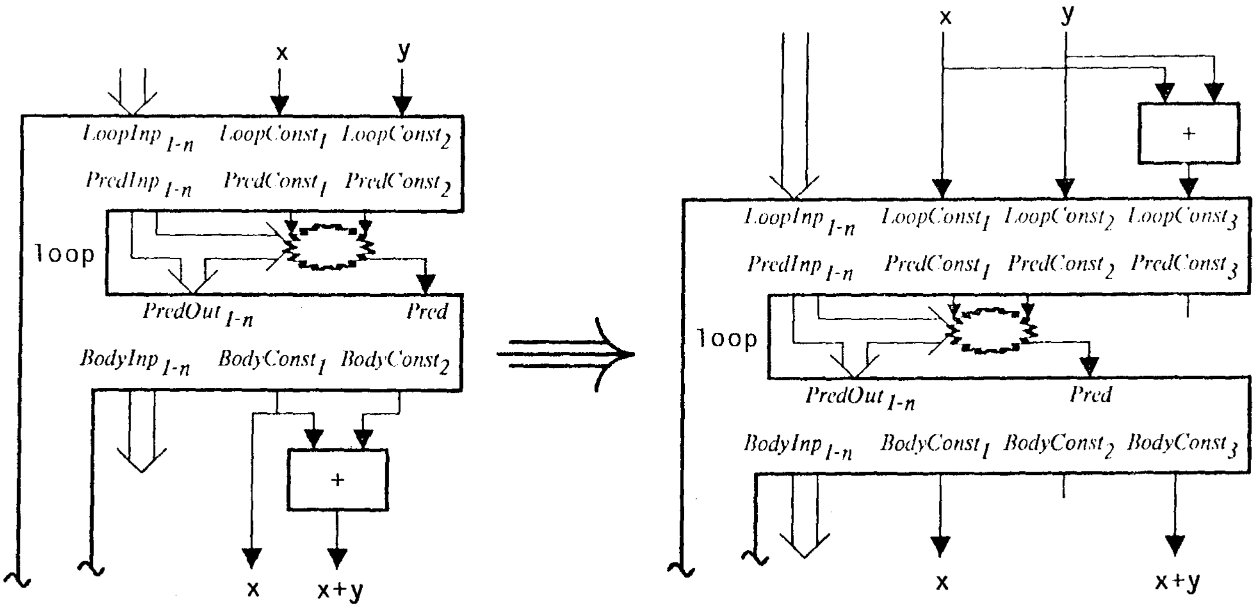
(We have introduced a new bit of notation here: an encircled value is shorthand for a `constant` instruction bearing that value.) The figure shows how motion across the bottom of an `if` can lead to I-fetch Elimination, as for the following Id fragment:

```
let
  p, q = if x < y then x, y else y, x;
  ...
```

While it is always safe to move code containing side-effects out of a conditional, provided identical code is moved out of both branches, it is not always safe to move such code into conditionals. Why? Because computation of the predicate may be contingent on the execution of those side effects, but neither side of the conditional can execute if the predicate has not been computed.

4.3.2 Code Motion Across *loops*

Code motion across `loop` encapsulators is mainly done to remove invariant subexpressions. The program graph form of loops makes it very easy to detect such expressions: a subgraph is a loop invariant expression if its inputs only come from *PredicateConstant* or *BodyConstant* ports of the `loop`. These subexpressions can be lifted outside the loop by adding additional *LoopConstant*, *PredicateConstant*, and *BodyConstant* ports, as illustrated below.



Care must be taken when an expression is lifted from the loop body, since after lifting it will always be executed, but before lifting it would not have executed if the initial evaluation of the predicate returned false. This might result in an arithmetic overflow or division by zero that otherwise would have been prevented. Happily, dataflow systems generally handle such errors by means of error tokens [Wetherell 82], so this would not crash the user's program. On the other hand, we might wish to avoid the computation if it is not necessary. In any case, we can choose to fix this bug by enclosing the lifted code in a conditional.

We also point out that if there is a direct connection between a *BodyInput* and a *BodyOutput*, and another direct connection between the corresponding *PredicateInput* and

PredicateOutput, then the circulating variable represented by that set of ports is in fact invariant. In that case, it can be converted to a loop constant, and will then be subject to motion as described above.

The lifting of loop invariants was also reported in [Skedzielewski 85a].

4.4 Interprocedural Side-Effects Analysis

Many of the program graph optimizations described earlier depend on knowing which instructions can cause side-effects. In the case of the `apply` instruction, that knowledge in turn depends on the properties of the procedure being applied, which depends on the properties of any procedures *it* applies, and so on. Here we examine some ways of determining which procedures can result in side-effects.

The analysis is fairly simple if we restrict ourselves to the first order case, in which every identifier which denotes a procedure is a constant, and is always applied to the correct number of arguments (*i.e.*, the same number of arguments as its arity) at once. In that case, we can define a function $\mathcal{SFF}$ from procedure identifiers to booleans such that if $\mathcal{SFF}(\mathtt{f})$ is false then application of `f` can never cause side effects, but if $\mathcal{SFF}(\mathtt{f})$ is true then such an application might cause a side effect.

$\mathcal{SFF}$ is computed in the following manner. We examine each procedure and note which procedure identifiers appear within its body and whether or not the body contains a side effect causing instruction (`array` or `i-store`). We then write a set of *n* equations, one for each procedure:

$$\mathcal{SFF}(\mathtt{f}_i) = \mathit{LocalSideEffects?}(\mathtt{f}_i) \vee \mathcal{SFF}(\mathtt{f}_{i,1}) \vee \mathcal{SFF}(\mathtt{f}_{i,2}) \vee \ldots$$

where $\mathit{LocalSideEffects?}(\mathtt{f}_i)$ is true if and only if $\mathtt{f}_i$ contains an `array` or *i-store* instruction, $\mathtt{f}_{i,1}$, $\mathtt{f}_{i,2}$, ... are those procedure identifiers which appear in $\mathtt{f}_i$'s body, and $\vee$ denotes the boolean inclusive or operator. We now have a set of mutually recursive equations over the two element domain **false** $\subseteq$ **true**. Since $\vee$ is monotonic and continuous over this domain, the equations have a solution (least fixpoint), which may be computed by a variety of methods such as Kleene recursion.

Extending the analysis to handle procedures passed as arguments and higher order procedures is a subject for future research.

5. The Tagged-Token Dataflow Architecture

The program graph concisely expresses the computation implied by an Id Nouveau program, but does little good unless run on a real machine. Translating from program graph to machine graph takes the program from the realm of the abstract to the realm of the concrete, as the machine graph is literally object code for a dataflow computer. In the program graph we appealed to intuition as we glossed over details of matching tokens with one another, of implementing unusual firing rules (*e.g.*, that for `constant`), and of managing the finite resources of a real machine. Our translation to machine graph must take all these into account.

To understand the machine graph, however, we must first become familiar with dataflow architectures and see just what constraints must be met. Our discussion will be based on the MIT Tagged-Token Dataflow Architecture [Arvind 83, Arvind 86b], but the features important to us are typical of most other dynamic dataflow architectures, such as the Manchester Machine [Gurd 85] or Sigma-1 [Hiraki 84]. Compilation for static dataflow architectures presents a somewhat different set of problems, which we shall not address. They are adequately described elsewhere [Ackerman 84]. Since our primary concern is compilation and not architecture, we will allow ourselves a little license in describing the tagged-token architecture; the interested reader is invited to read [Arvind 85] and [Arvind 86b] for more accurate information.

5.1 Machine Organization

A stylized block diagram of the tagged-token architecture is shown in Figure 5-1.

Instructions in the tagged-token dataflow architecture are restricted to having just one or two inputs, and, with one exception, only one output. A single firing rule suffices for all instructions: an instruction executes when all of its input tokens are available, removing the input tokens and sending a result token to each of the instructions to which its output is connected. In other words, all machine graph instructions behave pretty much like the program graph's + instruction. Instructions are grouped into *code blocks*, and are addressed by offset from the beginning of a block.

The *waiting-matching* unit in the machine is responsible for routing tokens to the *ALU*

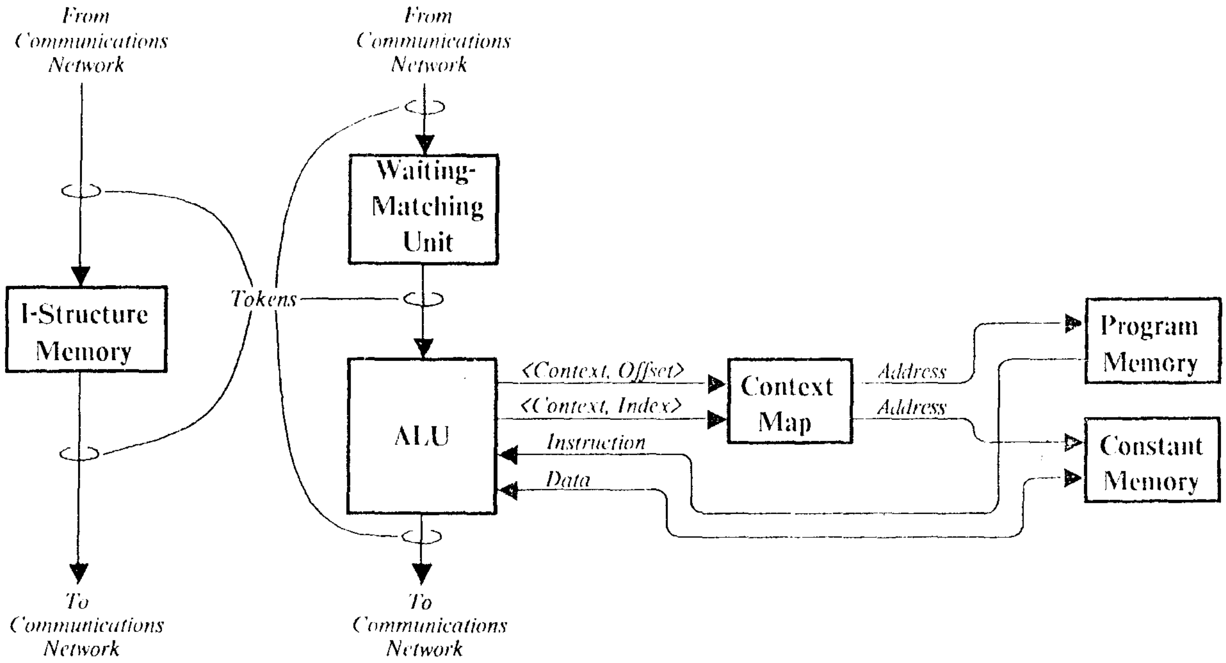
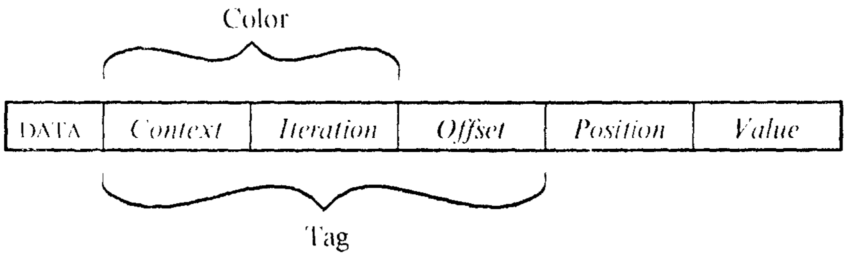


Figure 5-1: The Tagged-Token Dataflow Architecture (Simplified)

for execution. The waiting-matching unit must determine to which instruction a token is headed. If that instruction is a two-input instruction, then the waiting-matching unit must find that token's partner so that both can be presented to the ALU simultaneously. This is the origin of the unit's name: the first token to arrive for a two-input instruction *waits* for its partner to arrive and *match* with it. There may be many independent sets of tokens headed for the same instruction, either because a procedure is invoked more than once or because we allow several iterations of a loop to proceed in parallel.

Correct pairing of tokens is accomplished with a simple tagging scheme that puts some extra bits on each token. Tokens take the following form:



Each of the fields of a token is of fixed size, and their meanings are as follows:

Context Serves to distinguish between sets of tokens that are sharing the same code block; for example, two different invocations of a procedure. Two tokens are part of the same invocation if and only if their *context* fields are the same. A table indexed by context number associates a code block and a constant area with each context.

Iteration A refinement of the *context* field that serves to distinguish between sets of tokens representing different iterations of a loop.

Offset Indicates to which instruction within the code block indicated by the context the token is heading.

Position Indicates to which input of the instruction the token is heading if the instruction has two inputs.

Value The actual data carried by the token.

(This is somewhat simplified; in the real machine there are a few more bits to simplify routing and matching.) The context and iteration fields together are also called the *color*[4], and the context, iteration, and offset fields together are called the *tag*. The idea is that there are effectively many copies of each code block, with the color field indicating to which copy a token belongs. The tag field is important because two tokens are to be consumed together by a two-input instruction — they match — if and only if their tag fields are identical.

The ALU is responsible for execution of instructions. It has access to two kinds of

[4] Also called *hue*, or by some other chromatic name.

memory. *Program Memory* contains the dataflow graph itself, grouped into code blocks as mentioned earlier. *Constant Memory* serves as a scratchpad for holding things like loop constants; it is divided into regions called *Constant Areas*. The *Context Map* associates a code block and constant area with each context number; instructions and constant area elements are addressed by context number and offset. There are no restrictions on this mapping, and a common occurrence is to have several contexts map to separate constant areas but share the same code block.

Each instruction in a code block carries its opcode and a list of *destinations*, one destination for each input to which the instruction's output is connected. A destination contains the appropriate offset and position field for sending a token to a particular input of a particular instruction. Most instructions form output tokens by combining the offset and position from each destination with the context and iteration from the input tokens; thus, most instructions keep tokens within a given color. There are a small number of instructions which use different rules for constructing their outputs' tags, and these instructions are used for transporting tokens between contexts or iterations.

Most tokens in the machine follow the waiting-matching-ALU path, but there is another path through *I-structure memory*. As the name suggests, I-structure memory is responsible for maintaining and manipulating any I-structures used by a program. An I-structure location may be read by sending a special I-STR-FETCH token to I-structure memory. This token contains the address of the location to be fetched, and a context, iteration, offset, and position. If the desired location has already been written, the I-structure memory responds to the fetch request by sending back an ordinary token whose context, iteration, offset, and position fields are taken from the I-STR-FETCH token, and whose value field contains the value fetched from the I-structure. If the location had not yet been written when the fetch request was received, the request is recorded in a *deferred-read list* for that location. When the location is finally written, by sending I-structure memory a I-STR-STORE token containing the address and value to be stored, tokens are sent back for every deferred read in the deferred-read list, and subsequent reads proceed normally. To the program, there is no difference between a deferred read and a normal read, other than the time it takes for the result to arrive. On the other hand, the ALU is free to execute other instructions once it has sent the I-STR-FETCH token; it need not wait for the result to arrive. This accounts for the dataflow machine's ability to tolerate memory latency.

How does I-structure memory differ from constant area memory? For one thing, an individual constant area is accessible only to instructions executing in a particular context. This is clearly not suitable for I-structures as they exist in Id Nouveau, for I-structures may be passed far and wide among procedures, yet correct matching requires different context numbers be assigned to different invocations. Another difference: unlike I-structure memory, constant area has no built-in synchronization mechanism, and so we must arrange for our code never to read from constant area location until it is known that that location has been written. On the other hand, the proximity to the ALU and the absence of possibility of deferred reads allows constant area to be accessed as rapidly as program memory. This makes it suitable for holding loop constants, as long as we withhold loop execution until the constants are stored, and are able to detect termination of the loop so that the constant area can be reclaimed. We are relying here on the known lifetime of loop constants.

Figure 5-1 shows only one I-structure memory and only one processing element (i.e., waiting-matching-ALU path). Of course, a complete dataflow machine is composed of many such I-structure memories and processing elements, in equal or unequal numbers as desired. This assumes some mapping scheme for distributing I-structure addresses among the various I-structure memories and context numbers among the various processing elements. Developing effective mapping schemes is a topic of current research and will not be considered here.

Even though there are many I-structure memories and many processing elements, there is logically a single agent for allocating I-structures and contexts. We say "logically" because each request to allocate an I-structure or context must receive a globally unique answer. On the other hand, we are not ruling out a distributed implementation of this agent, able to service many requests simultaneously. In fact, we will treat this agent as a black box, called the "Manager", without saying whether its implementation is localized or distributed, in hardware or in software. Parenthetically we note that even a dataflow implementation of the manager is not impossible; see [Arvind 84] for details.

5.2 Implications for Machine Code

A realistic dataflow architecture such as the tagged-token architecture just described is powerful, but incapable of directly executing the program graphs we have presented. Its restrictions give rise to the following considerations in translation to machine code.

1) Machine instructions are limited to two inputs and one output, and have a fixed firing rule. Complex program graph instructions like `if` and `loop` will have to be implemented by collections of machine graph instructions.

2) Instructions must execute in a constant (and hopefully small) amount of time, precluding the use of instructions that do unbounded computation or waiting.

3) Certain program graph instructions have "intelligent" firing rules, an example being the `constant` instruction which emits a token whenever needed. A more causal implementation must be found for the machine graph.

4) The tagging mechanism must be brought to bear on the problem of keeping independent sets of tokens from being confused.

5) The finite number of context numbers requires the reclamation of contexts when no longer in use. This in turn requires the ability to detect termination of regions of code.

6) The finite number of iteration numbers requires clever control of loops to prevent exhausting this resource while still allowing a sufficient amount of parallelism.

7) The operational semantics implied by the program graph's firing rules must be preserved.

Careful attention to these points, especially numbers 3, 5, and 6, are what separate a hypothetical implementation from a practical one.

6. Triggers and Signals

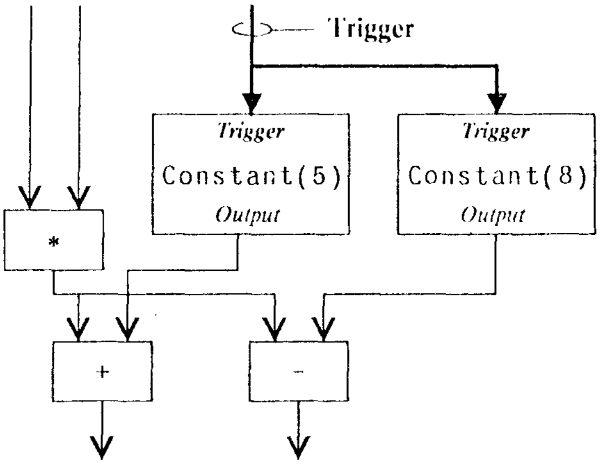
In the last chapter we noted that in the machine graph all instructions must fire because they receive some input, but that this is not quite true of the program graph because of instructions like `constant`. Furthermore, it was noted that we must be able to detect when all instructions in a given region of code have fired. We meet these two requirements through *triggers* and *signals*.

Triggers are extra arcs added to a dataflow graph to make sure that all instructions that are supposed to fire do in fact fire. Signals are extra arcs added to facilitate the detection of termination. Both are necessary features of the machine graph. As it turns out, however, triggers and signals make as much sense in the program graph as they do in the machine graph. Moreover, it is far easier to introduce them into the program graph, where more of the structure of the original Id Nouveau program is preserved. In this chapter we describe how a program graph without triggers and signals is transformed into an equivalent program with triggers and signals. The resulting graph is called a *well-connected program graph*, and its construction is the first step in the conversion to machine graph.

6.1 Triggers

Our program graphs contain a few instructions, such as `constant`, whose firing rules require them to produce an output "whenever needed". In the machine graph, of course, these instructions will have to be fired by the arrival of a "trigger" token, which indicates that the output is in fact needed. The `constant(5)` machine graph instruction, therefore, emits a token carrying `5` whenever it receives an input token. The value on the input token is ignored.

In a basic block, it is known that each instruction must execute exactly once whenever a set of tokens appears at the block's inputs. Therefore, we conclude that any instructions in the block which require triggers should each receive one trigger token when inputs arrive for the block. The algorithm for adding triggers to basic blocks is simply to add a new input to the block and wire it to all instructions in the block that require triggers; that is, to any unconnected input of an instruction. It is up to the encapsulator that encloses the block to provide a token for the new trigger input.[5]



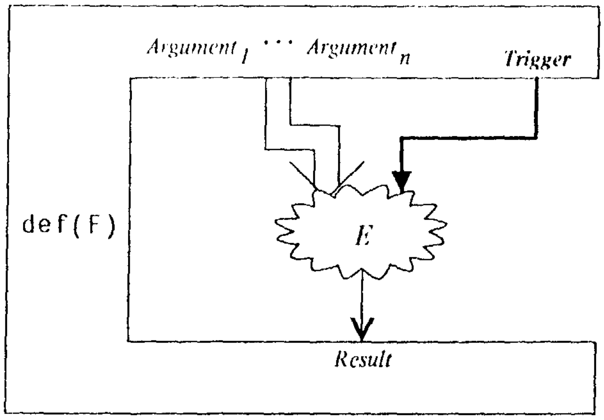
We now know how to add triggers to basic block, and so we now must show how triggers are propagated across encapsulators. The simplest encapsulator is the `def` instruction. The `def` instruction must provide a trigger to its enclosed block as soon as the procedure which it represents is invoked. Because our procedures are non-strict, however, we are not guaranteed that any of the argument tokens are available at invoke time. This precludes deriving the trigger as a function of any of the arguments, as for example by always using the first argument as the trigger. Instead, we change the definition of `def` so that in addition to its *Argument* outputs, it also has a special *Trigger* output which emits a token whenever the procedure is invoked. The value of this token is unimportant since it will only be used as a trigger.

——— **Output** ***Trigger*** **of** `def` ———

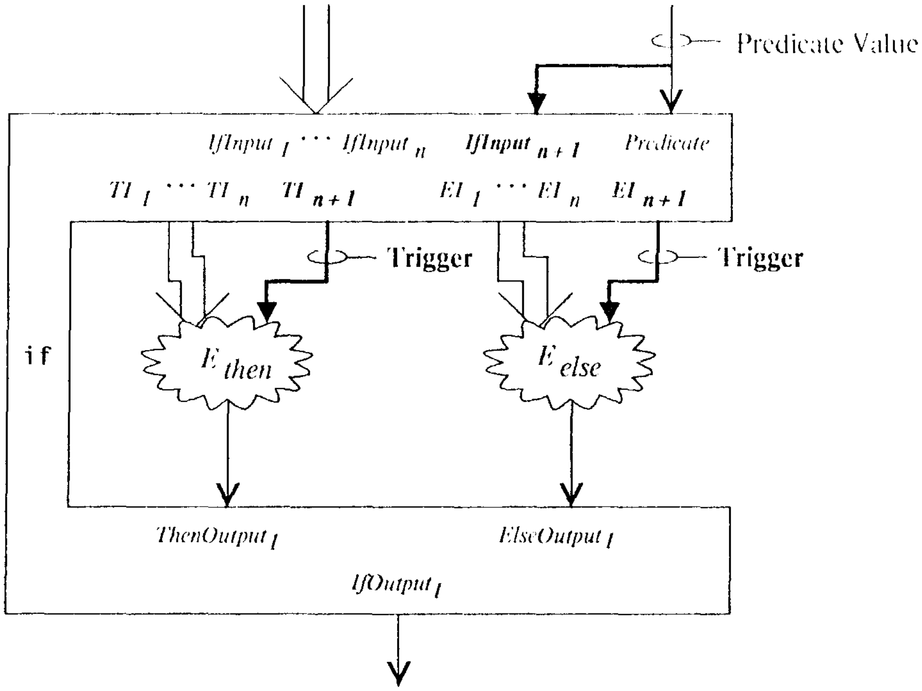
Pre-Condition: Procedure invoked. **Value Produced:** Anything

Our scheme for translating `def` into machine code must now be sure to provide this trigger token. The rule for triggering the enclosed block of a `def` is expressed in the following figure:

[5] This is a simplification; as we will see in Chapter 8, we will sometimes obtain the trigger for `constant` instructions from a different source.

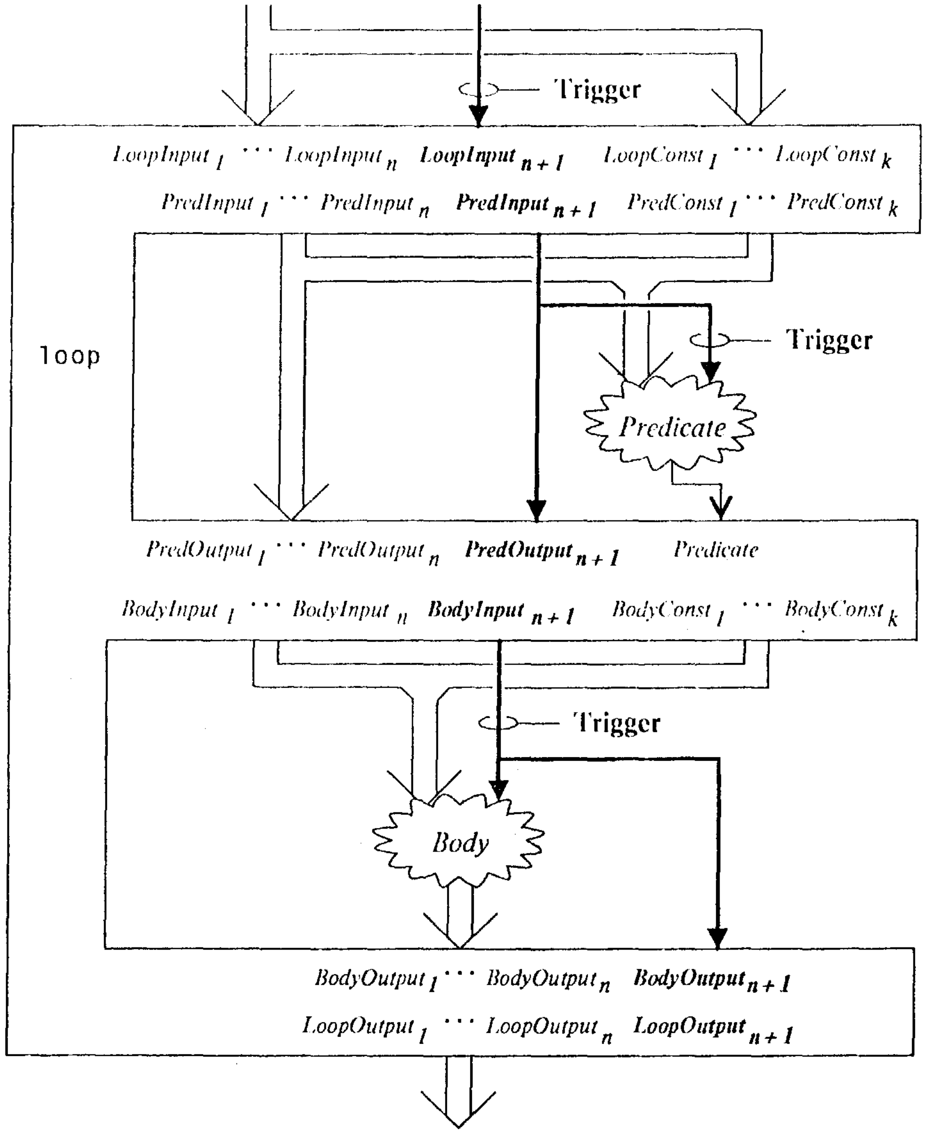


The `if` instruction contains two basic blocks, the `then` block and the `else` block. A trigger must be provided to the appropriate side as soon as it is known which block will execute. Again, it is not safe to use an existing *ThenInput* or *ElseInput* as the trigger, since the arrival of any or all of these may depend on instructions within the `if` being triggered. Instead, we add an additional *IfInput*, *ThenInput*, and *ElseInput* port to the `if` specifically for the trigger, as shown below.



Since the trigger is derived from the predicate, it is delivered to the `then` or `else` side as soon as that side is selected, regardless of which other inputs to the `if` have arrived. Notice that in the figure we have not shown a trigger for the predicate expression; if it requires one, it will be added when the block in which it is enclosed is processed.

The last case to consider is the `loop` instruction. Here we need to provide a trigger to the predicate block every time the predicate is to execute, and to the body block every time the body is to execute. One easy way to accomplish this is to circulate a trigger around the loop:


Trigger
$LoopInput_1 \cdots LoopInput_n$ $LoopInput_{n+1}$ $LoopConst_1 \cdots LoopConst_k$
$PredInput_1 \cdots PredInput_n$ $PredInput_{n+1}$ $PredConst_1 \cdots PredConst_k$
loop
Trigger
Predicate
$PredOutput_1 \cdots PredOutput_n$ $PredOutput_{n+1}$ Predicate
$BodyInput_1 \cdots BodyInput_n$ $BodyInput_{n+1}$ $BodyConst_1 \cdots BodyConst_k$
Trigger
Body
$BodyOutput_1 \cdots BodyOutput_n$ $BodyOutput_{n+1}$
$LoopOutput_1 \cdots LoopOutput_n$ $LoopOutput_{n+1}$

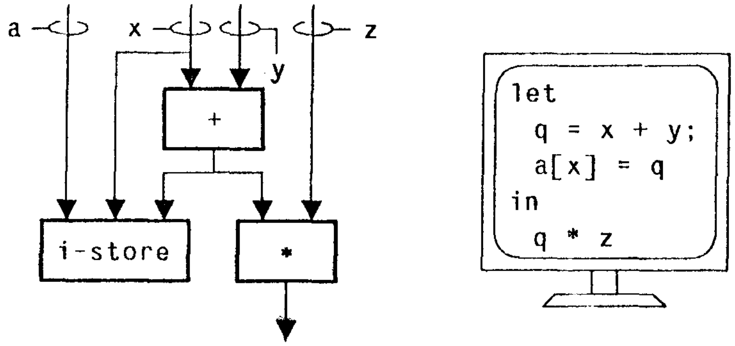
The initial token for this circulating trigger is the trigger for the block of which the `loop` is a part; that way, we are assured of receiving a trigger for the first execution of the predicate.

We can eliminate the overhead of an additional circulating variable, however, by exploiting the properties of the `loop` instruction. Specifically, we recall that the definition of loops is such that no predicate or body instruction need execute until all predicate or body inputs are present. This is in contrast to `if`, where we must be prepared to execute instructions in the `then` or `else` blocks even if some of their inputs are not yet available. Since in the loop we know that all inputs to a block will be present before we have to start executing the block, we can choose any of the existing circulating variables to serve as the trigger without changing the meaning of the program. In practice we want to overlap some number of iterations, and so it is advantageous to choose a circulating variable that we believe will be computed faster than any of the others, such as the index of a `for` loop. Then again, to increase exposed parallelism even further we may wish to retain the separate circulating trigger, thus triggering things as fast as possible.

To summarize, the algorithm for trigger addition is to begin with the innermost basic blocks, adding trigger arcs if needed. These triggers are then connected to the encapsulators enclosing their blocks, and the algorithm is repeated on the next innermost set of basic blocks. The process is complete when the outermost encapsulator (the `def`) is processed.

6.2 Signals

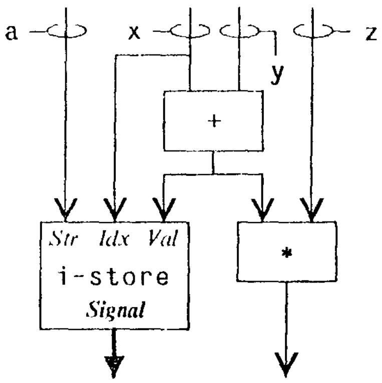
Whereas triggers are concerned with making sure all instructions are capable of firing, signals are concerned with ascertaining when all instructions have indeed fired. The definition of basic blocks guarantees that if a token arrives at every input of a block, then eventually every instruction in the block will execute and a token will be produced at every output of the block. Unfortunately, just because a token has appeared at every output does not mean that all instructions in the block have executed. In the following basic block, for example, a token may appear at the output even though the `i-store` instruction has not yet fired.



```
let
  q = x + y;
  a[x] = q
in
  q * z
```

We would like to add additional outputs to basic blocks such that if a token arrives at all outputs of the block, including the additional outputs, then every instruction in the block has fired. These additional outputs are called *signals*. Keep in mind that the signals do not imply termination by themselves, but only when accompanied by the other, non-signal, outputs.

No signals are needed to detect the execution of side-effect free instructions such as +, for they already produce a token as evidence of their execution. On the other hand, when an instruction acts only through side-effect, as does `i-store`, there is no token produced to indicate that the side-effect has taken place. So we must modify these instructions to produce a signal token when they have completed their side-effect. Our example then becomes:



The data carried on the signal token is unimportant. The new firing rules for `apply` are:

——— **Side-Effect of** `i-store` ———

Pre-Condition: *Structure*, *Index*, and *Value* present.

Effect: If element *Index* of *Structure* was not yet written, *Value* is written there, otherwise an error flag is raised.

——— **Output** ***Signal*** **of** `i-store` ———

Pre-Condition: *Structure*[*Index*] written.

Value Produced: [Anything]

The only other program graph instruction that needs a signal output is `apply`. We need a signal out of `apply` in addition to its regular output for two reasons. First, the regular output can be produced even if no token has yet been received on input *Argument*, and so without the signal we do not know whether the instructions that compute *Argument* have fired. Second, if the `apply` actually caused the invocation of a procedure (as opposed to the creation of a closure), then we are interested in detecting when the invoked procedure terminates. The appearance of a token at the regular output does not imply that all instructions in the invoked procedure have executed, for the same reason that a data value appearing at the output of a basic block does not imply that all its instructions have executed. apply, therefore, produces a signal when the argument is received and stored in the closure or sent to the invoked procedure and the invoked procedure terminates. The updated firing rule for apply is:

——— **Output** *Output* **of** apply ———

Pre-Condition: *Procedure* present.

Value Produced: Result of applying *Procedure* to *Argument*

——— **Output** *Signal* **of** apply ———

Pre-Condition: *Procedure* and *Argument* present, and *Argument* written to closure or sent to invoked procedure, which must have terminated.

Value Produced: [Anything]

Given these modifications to program graph instructions, we define the signals for a basic block as all of the unconnected outputs of instructions within the block. This includes the signal outputs of i-stores and applys, as well as unused outputs of ifs and loops. The latter might arise, for example, if a newified variable of a loop is not used in the return expression.

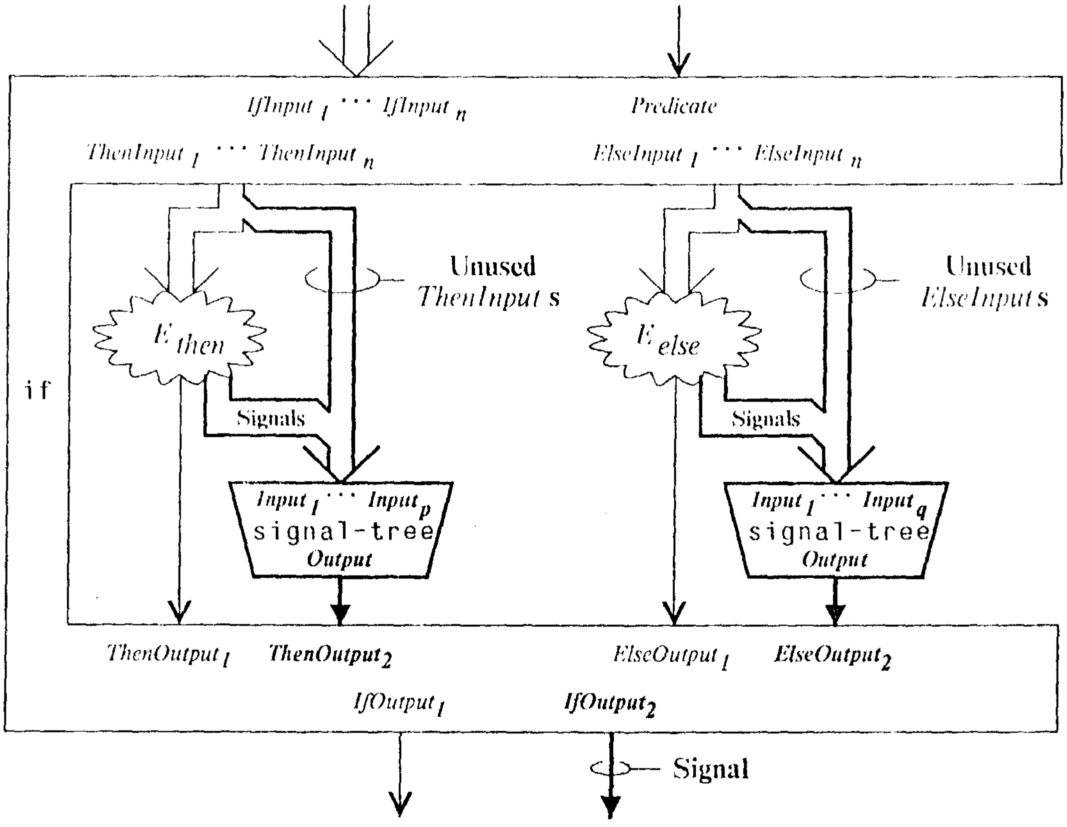
Given the rules for constructing the signals of basic blocks, we now examine how they propagate across encapsulators. An if instruction encloses two basic blocks, exactly one of which is executed once each time the block enclosing the if instruction executes. So if the predicate is true, for example, the then block executes, possibly producing a set of signals. But there is an additional source of signals for the then side besides those of the then basic block. Any *ThenInput* outputs of the if that are not connected to the then block (because the variables they represent are needed only by the else block, for example) are also signals for the then side, since there is no other way to detect that the instructions feeding the corresponding *IfInput* have executed. The analogous situation holds for the else side.

With the signals for each side of the if at hand, it is necessary to propagate them across the bottom of the if. First, we collect the signals from each side into a single signal per side by wiring each set to a signal-tree instruction. A signal-tree instruction simply produces a token when it receives a token on each of its inputs:

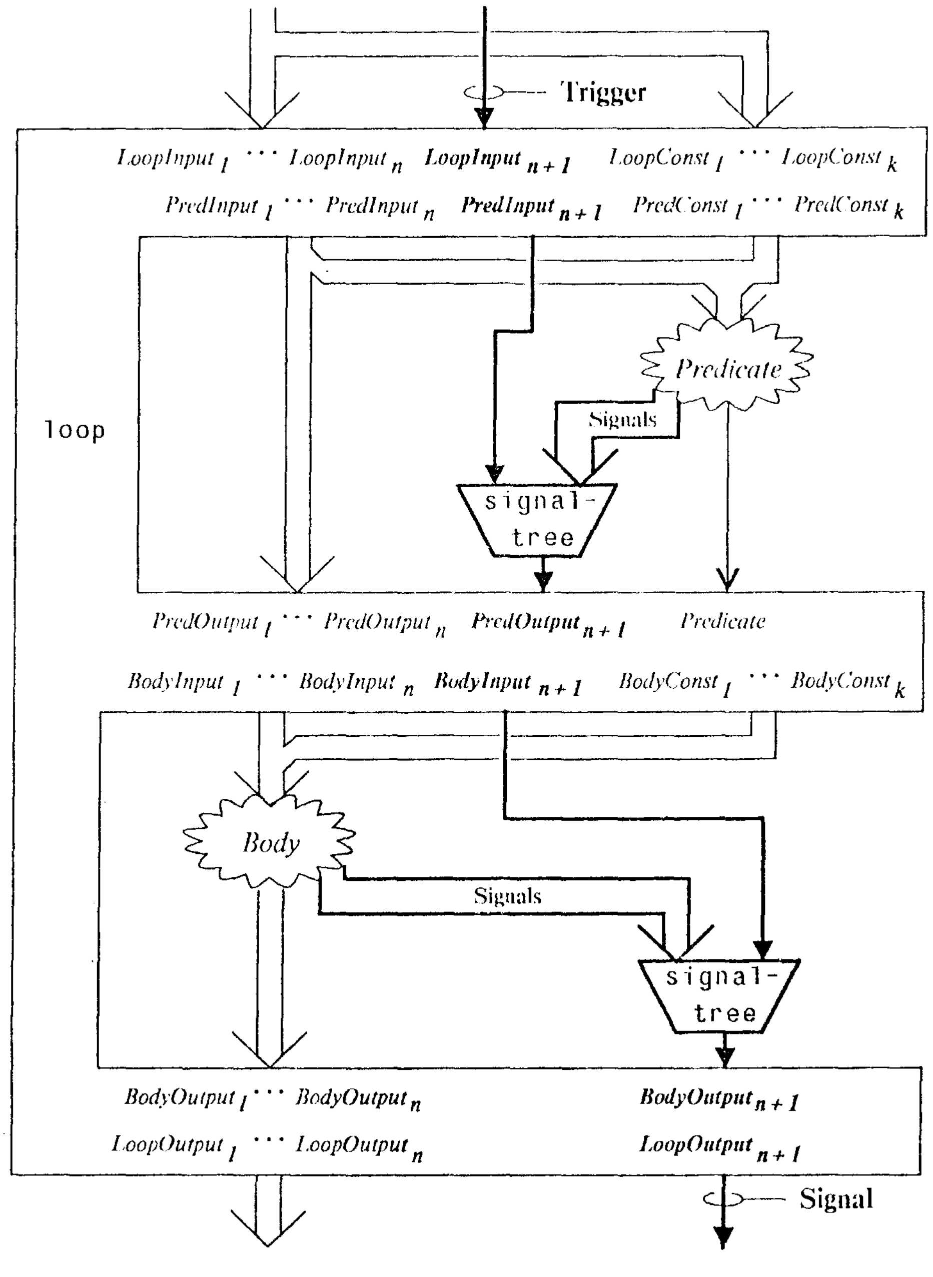
——— **Output** *Output* **of** `signal-tree` ———

Pre-Condition: All *Input*$_i$ present **Value Produced:** [Anything]

Next, we add another *IfOutput* to the `if`, along with a corresponding *ThenOutput* and *ElseOutput*. Finally, we wire the output of the `then`'s `signal-tree` instruction to the new *ThenOutput*, and the `else`'s `signal-tree` to the new *ElseOutput*. The new *IfOutput* becomes a signal for the block enclosing the `if`, since it is now an unconnected output (recall that signal and trigger generation proceeds from the innermost blocks outward). All of this is summarized in the figure below.

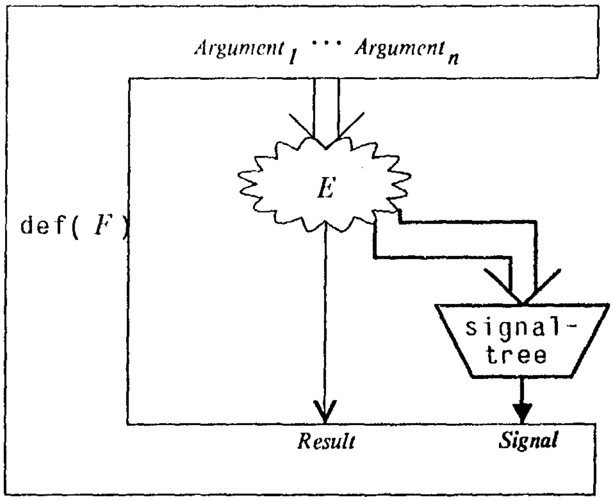


In `loop` instructions, either the predicate block or the body block can produce signals. If either produces signals, we must collect those signals for all iterations, so that the final set of tokens emitted from the loop implies that all executions of the encapsulated blocks have finished. We do this by creating a circulating signal, which essentially acts as a signal for all previous iterations. By combining it with the signals from the current iteration, we obtain its value for the next iteration. This is depicted in the figure below.


Trigger
LoopInput 1 ··· LoopInput n LoopInput n+1 LoopConst 1 ··· LoopConst k
PredInput 1 ··· PredInput n PredInput n+1 PredConst 1 ··· PredConst k
loop
Predicate
Signals
signal-tree
PredOutput 1 ··· PredOutput n PredOutput n+1 Predicate
BodyInput 1 ··· BodyInput n BodyInput n+1 BodyConst 1 ··· BodyConst k
Body
Signals
signal-tree
BodyOutput 1 ··· BodyOutput n BodyOutput n+1
LoopOutput 1 ··· LoopOutput n LoopOutput n+1
Signal

The initial token for the circulating signal is the trigger for the enclosing block, and the final value of the circulating signal becomes a signal for the enclosing block by virtue of the corresponding *LoopOutput* being unconnected. As with `ifs`, there are two sources of signals for the predicate and body: the signals from the basic block itself as well as any unconnected *PredicateInput* or *BodyInput* outputs[6]. An important property of this signalling scheme is that when all *PredicateOutput* or *BodyOutput* inputs have received a token for a given iteration, all instructions in the predicate or body for that iteration have executed. Our signal, therefore, serves as the signal for individual iterations as well as the signal for the entire `loop` instruction. This is critical to the recycling of iteration identifiers.

Finally, we take care of signals for the body of `def` instructions. The `def` instruction is responsible for sending a signal back to the `apply` that invoked it to indicate that all instructions within the `def` have executed. We therefore collect all signals from the `def`'s body into a single signal, which we wire to a special *Signal* input of the `def`.



Signal addition, like trigger addition, proceeds from the innermost basic blocks outward, since the addition of signals to an inner block can create a signal in the enclosing block. Trigger addition must precede signal addition, since the former can introduce unconnected outputs which become signals (for example, if only one side of an `if` needed a trigger, an unconnected output is created on the other side). Rather than compute all triggers and then all signals, it is

[6] Our schema for `while` expressions, however, will never leave unconnected *PredicateInput* ports since each one is wired to the corresponding *PredicateOutput* port regardless of whether it is used by the predicate.

7. The Machine Graph

Now we show how to systematically translate a program graph into a machine graph which contains only instructions suitable for execution on the tagged-token dataflow architecture and which takes into account the finite resources of this machine. Once we have added signals and triggers to the program graph to obtain a well-connected program graph, translation to machine code is accomplished by substituting groups of machine instructions for each of the program graph instructions. This phase is therefore a kind of graphical macro expansion.

7.1 Instruction Set

We can describe the instruction set of the dataflow machine by expressing the output token of each instruction in terms of its input tokens. Remember that all instructions have the same firing rule, namely, an instruction fires when all of its inputs have received tokens bearing the same tag.

We will use the following notation for tokens:

<DATA, *context*, *iteration*, *offset*, *position*, *value*>
An "ordinary" token, as what normally flows along arcs of the dataflow graph. The meanings of the fields were described earlier.

<I-STR-FETCH, *address*, *context*, *iteration*, *offset*, *position*>
A request to fetch the contents of I-structure location *address.*

<I-STR-STORE, *address*, *value*>
A request to store *value* into I-structure location *address.*

A typical two-input instruction is +, which we describe as follows:

——— **Machine Instruction +** ———

Inputs:
<DATA, c, i, q, 1, v_1>
<DATA, c, i, q, 2, v_2>

Output:
<DATA, c, i, Dest(q), Pos(q), $v_1 + v_2$>

Dest(q) and Pos(q) refer to the instruction offset and input position in the destination list of instruction q; output tokens are sent for each offset/position pair in the destination list.

Notice that both input tokens have the same value for c, i, and q, as this is a consequence of the firing rule.

An example of a one-input instruction is `identity`:

——— **Machine Instruction** `identity` ———

Input:	**Output:**
⟨DATA, c, i, q, 1, v⟩	⟨DATA, c, i, Dest(q), Pos(q), v⟩

As we describe the translation to machine code, we will introduce additional machine instructions as needed.

7.2 Basic Program Graph Translations

The arithmetic, relational, and logical program graph instructions (`+`, `<=`, `or`, *etc.*) have exact equivalents in the machine instruction set, since they each have two inputs and one output. Their definitions are analogous to that of + given in the previous section.

Similarly, the `constant` instructions as found in the well-connected program graph are machine instructions, as they have one input (the trigger) and one output. The definition of a typical constant instruction is:

——— **Machine Instruction** `constant(5)` ———

Input:	**Output:**
⟨DATA, c, i, q, 1, v⟩	⟨DATA, c, i, Dest(q), Pos(q), `5`⟩

In the MIT tagged-token architecture, the instruction set actually allows constants to be made part of other instructions, *e.g.*, a + instruction whose second input is always the constant `1` is represented as a single instruction. We will defer discussion of this feature until the next chapter.

The `i-fetch` program graph instruction is also a primitive machine instruction, as it fits the two-input paradigm. It operates in two steps, however, owing to the nature of I-structure memory. The actual `i-fetch` ALU instruction operates as follows:

——— **Machine Instruction** `i-fetch` ———

Inputs:
⟨DATA, c, i, q, 1, S⟩
⟨DATA, c, i, q, 2, idx⟩

Output:
⟨I-STR-FETCH, Org(S) + idx, c, i, Dest(q), Pos(q)⟩

Here S is an *I-structure descriptor*, and Org(S) refers to the address in I-structure memory of the first element of S. All the ALU does is compute the address of the desired location by adding the given index to the origin of the given I-structure, and send a request token to the I-structure memory. When the I-structure memory is able to fetch the location, it responds with an ordinary token:

——— **I-Structure Operation** I-STR-FETCH ———

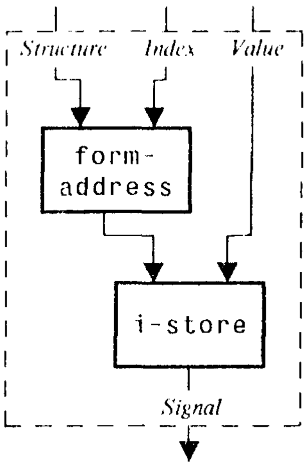
Input:
⟨I-STR-FETCH, $addr$, c, i, q, p⟩

Output:
⟨DATA, c, i, q, p, Contents($addr$)⟩

The net effect of an `i-fetch` is like that of an ordinary two-input instruction: two DATA tokens are consumed, and a DATA token carrying the result is produced. But while the entire operation may take an arbitrary amount of time depending on when the corresponding store happens, the ALU's role of sending the I-STR-FETCH instruction takes constant time.

The `i-store` instruction is similar to `i-fetch`: it sends a special I-STR-STORE token to the I-structure memory. It also needs to send a signal token for termination detection. One might expect that the signal is generated by I-structure memory upon completion of the store. In fact, we are generally not interested in making sure the store has taken place, but only that the `i-store` instruction itself has fired, and that the I-STR-STORE token is on its way. Hence, the signal is generated immediately by the ALU.

The `i-store` program graph instruction has three inputs: an I-structure, a subscript, and a value. Because machine instructions are limited to two inputs, we must use two machine instructions to implement `i-store`, one called `form-address` which computes the address of the I-structure location from the I-structure descriptor and the index, and one called `i-store` which forwards the address and the value to the I-structure memory and generates the signal.



——— **Machine Instruction** `form-address` ———

Inputs:
<DATA, *c*, *i*, *q*, 1, *S*>
<DATA, *c*, *i*, *q*, 2, *idx*>

Output:
<DATA, *c*, *i*, Dest(*q*), Pos(*q*), Org(*S*) + *idx*>

——— **Machine Instruction** `i-store` ———

Inputs:
<DATA, *c*, *i*, *q*, 1, *addr*>
<DATA, *c*, *i*, *q*, 2, *v*>

Outputs:
<I-STR-STORE, *addr*, *v*>
<DATA, *c*, *i*, Dest(*q*), Pos(*q*), [Anything]>

The implementation of the `array` instruction must invoke the resource manager. We can imagine a `allocate-array` machine instruction for that purpose; its two-step operation is not unlike that of `i-fetch`.

——— **Machine Instruction** `allocate-array` ———

Inputs:
<DATA, *c*, *i*, *q*, 1, *lb*>
<DATA, *c*, *i*, *q*, 2, *ub*>

Output:
<MGR-ALLOC, *lb*, *ub*, *c*, *i*, Dest(*q*), Pos(*q*)>

The token emitted by this instruction goes not to the processing element but to the manager, which allocates the token and sends the descriptor back:

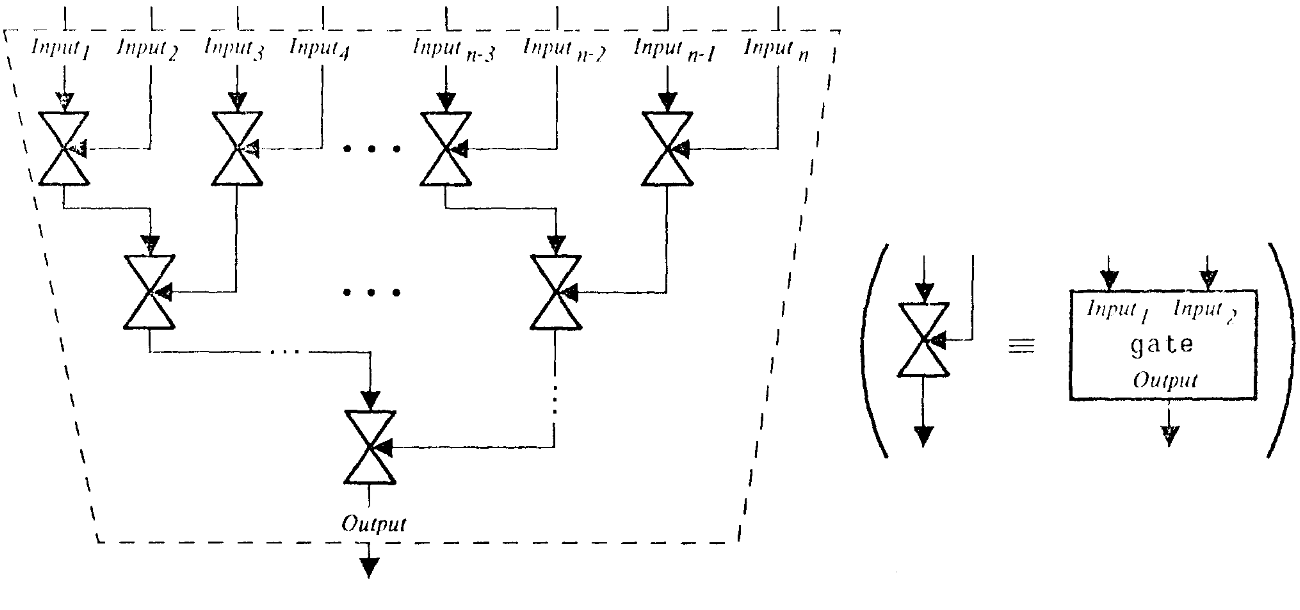
——— **Manager Operation** MGR-ALLOC ———

Input:
<MGR-ALLOC, *lb*, *ub*, *c*, *i*, *q*, *p*>

Output:
<DATA, *c*, *i*, *q*, *p*, *S*>

Here, S is the descriptor for the newly allocated I-structure which has lower bound lb and upper bound ub. As with `i-fetch`, the ALU's role of forwarding the request takes constant time even though the manager may take a long time to respond with the result. While this scheme for implementing `array` is completely general, we are not ruling out more sophisticated methods that avoid having to contact the global manager on every allocation. Such other methods may entail expanding the program graph `apply` instruction into a group of several machine instructions.

We implement `signal-tree` instructions by a tree of machine instructions called `gate`.



The `gate` instruction is sort of a two-input version of `identity`: it passes its first input unchanged, but only after both its first and its second inputs have arrived. The value of the second input is ignored.

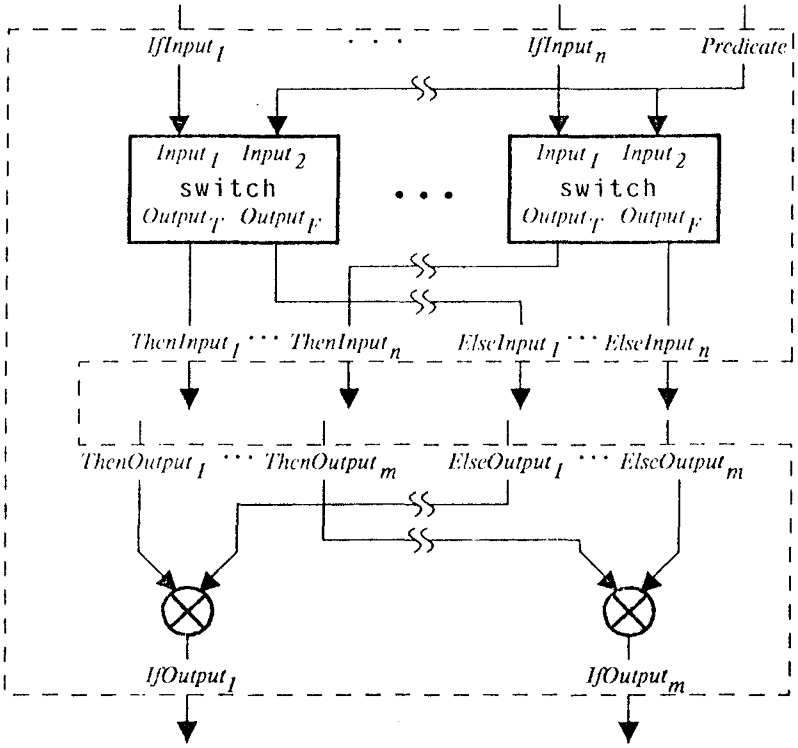
——— **Machine Instruction** `gate` ———

Inputs:	**Output:**
$\langle \text{DATA}, c, i, q, 1, v_1\rangle$ $\langle \text{DATA}, c, i, q, 2, v_2\rangle$	$\langle \text{DATA}, c, i, \text{Dest}(q), \text{Pos}(q), v_1\rangle$

Our translation for signal trees implies that the signal produced at the output will carry the same value as the leftmost input. This is of little consequence, since signals are always "don't care" values.

7.3 Translation of *if*

The translation of `if` is as follows:



We have used two unusual machine instructions, `switch`, and the non-deterministic merge, which is indicated by a circle and cross. The `switch` instruction is unusual because it has not one but two outputs; this instruction is the sole exception to the rule that all machine instructions have one output. `switch` instructions must therefore have two destination lists instead of just one. The definition of `switch` is as expected:

——— **Machine Instruction** `switch` ———

Inputs:

<DATA, c, i, q, 1, v>
<DATA, c, i, q, 2, `true`>

Output:

<DATA, c, i, $\text{Dest}_{\text{true}}(i)$, $\text{Pos}_{\text{true}}(q)$, v>

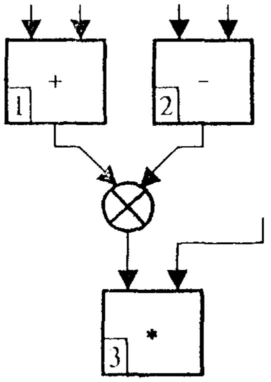
——— **Machine Instruction** switch ———

Inputs:
<DATA, c, i, q, 1, v>
<DATA, c, i, q, 2, false>

Output:
<DATA, c, i, $\text{Dest}_{\text{false}}(i)$, $\text{Pos}_{\text{false}}(q)$, v>

We have shown two rules for switch, one for when the control input is true, and one for when it is false. The only difference is in which destination list is used for generating the output tokens.

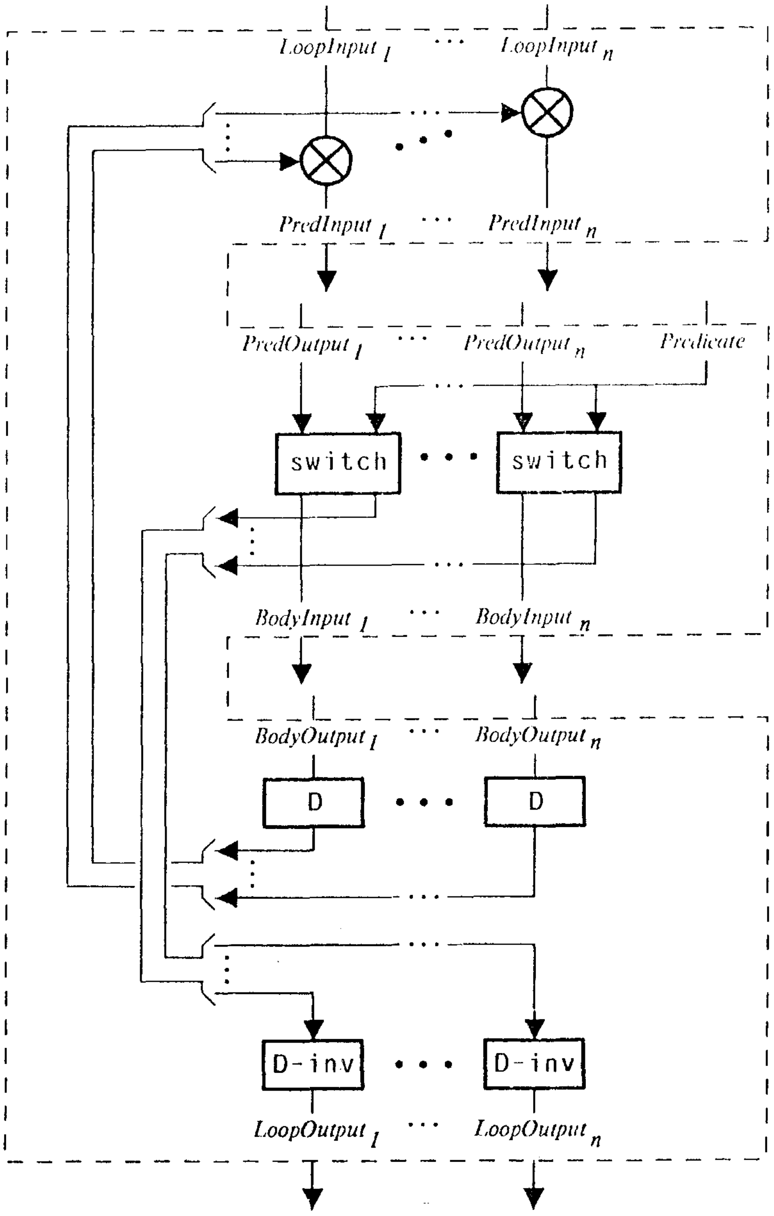
The non-deterministic merge takes a token from *either* input and passes it to its output. This is different than a typical two-input instruction, which waits for *both* inputs and computes its output as a function of the two. In fact, the merge is not really an instruction at all, but is implemented by playing with the destination lists of the instructions that feed it. For example, if we had the following machine graph fragment:



There would be no merge instruction in the code block at all; rather, the destination lists of both instructions 1 and 2 would have a destination indicating $Input_l$ of instruction 3. Hence, when either instruction 1 or instruction 2 fired, instruction 3 would receive a token on $Input_l$. We must insure, of course, that we never send a tokens with the same tag to *both* sides of the merge, and our translation for if satisfies this requirement.

7.4 Translation of *loop*

The loop instruction is complicated, and has a complicated translation. First, let us assume that the loop has no loop constants, and that the iteration field of tags is infinite. In that case, a loop would be translated as:



The initial tokens fall in through the merges to the row of switches and to the predicate.

The predicate decides whether the loop body is to execute. If so, then the predicate produces the value true, and the switches allow the loop variables to proceed to the loop body. As they leave the loop body, they pass through D instructions. A D instruction is a one-input instruction that increments the iteration field of the input tag[7]:

——— **Machine Instruction** D ———

Input: <DATA, c, i, q, 1, v>

Output: <DATA, c, $i + 1$, Dest(q), Pos(q), v>

In this way, the tokens for the next iteration will have different tags than for previous iterations, and will therefore not be confused even if we allow several iterations to proceed concurrently. After passing through the D instructions they pass through the merges, and the entire process repeats. When the predicate finally returns false, the latest values for the loop variables will exit the loop after passing through D-inverse instructions. A D-inverse instruction, as the name suggests, undoes the effects of D instructions:

——— **Machine Instruction** D-inverse ———

Input: <DATA, c, i, q, 1, v>

Output: <DATA, c, 0, Dest(q), Pos(q), v>

Why must we reset the iteration field of tokens leaving the loop? Because they will be used in computations involving tokens that were produced outside the loop. Only if the tokens leaving the loop have zero in their iteration field will they correctly match with the tokens outside the loop.

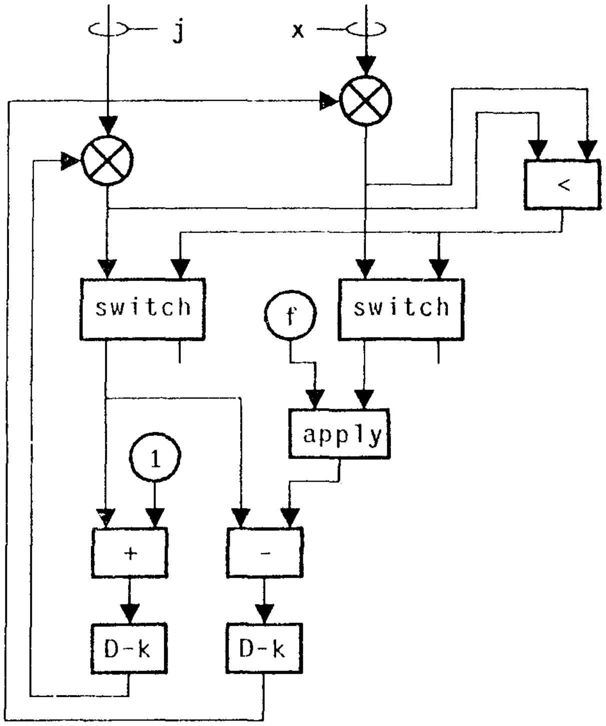
In the previous scheme, the iteration field on tokens could grow arbitrarily large: if the loop body executed n times, then the final set of tokens had n in their iteration field. In a practical machine, unfortunately, the iteration field must be of fixed size, requiring us to recycle iteration numbers. We can easily keep the iteration numbers within bounds by making the D instructions increment the iteration number modulo K, where K is the number of possible iteration numbers we wish to accommodate. We will call such a D instruction a D-K instruction.

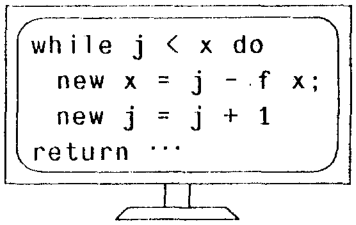
[7] The name D is used for historical reasons.

——— **Machine Instruction** D-K ———

Input:	**Output:**
⟨DATA, c, i, q, 1, v⟩	⟨DATA, c, $(i + 1) \bmod K$, Dest(q), Pos(q), v⟩

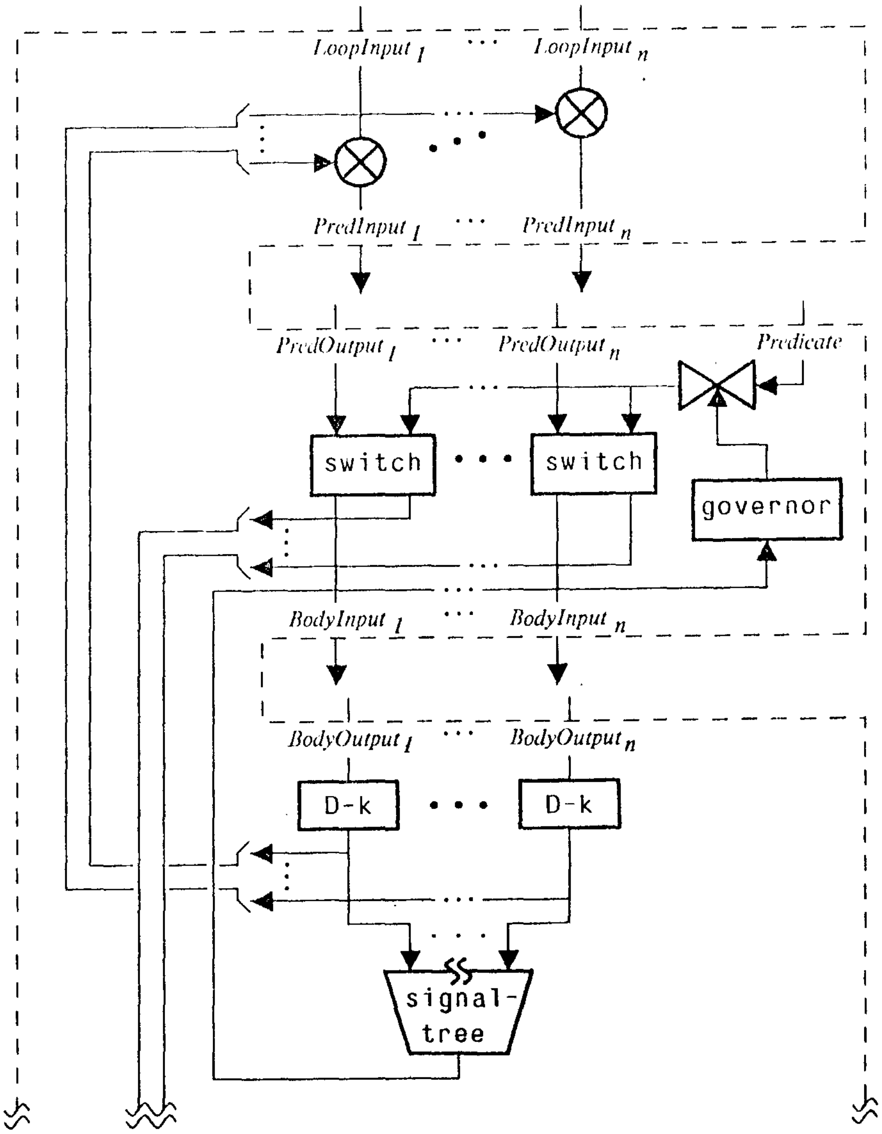
But replacing Ds by D-Ks causes other problems, as illustrated by the following program:




```
while j < x do
  new x = j - f x;
  new j = j + 1
return ···
```

(The triggers for the constant instructions and the signal output from the application of f have been omitted for clarity.) Suppose that the + executes quickly, but that procedure f takes a long time. The initial tokens for j and x would enter the loop body with iteration field zero. While f is computing the first value of (f x) the - instruction cannot execute, since it depends on the value of (f x). On the other hand, nothing prevents new values of j from being computed, and so newer and newer values for j will rapidly be injected into the loop body, where they accumulate at the input of the -. Each of these values of j will carry a different iteration field — that is, until K iterations have unfolded and the D-K instructions start recycling the iteration numbers. At that time, more than one token with the same tag appears at the input of the -, and the data for separate iterations is confused.

If we want to have only K possible values for the iteration number, then we must restrict the loop so that at most K iterations are in progress at a time. The idea is that we prevent an iteration from proceeding until we are sure that there exist no tokens from another iteration with the same iteration number. We can do this without peril to the our translation's correctness, for we reserved the right to execute the loop one iteration at a time, if necessary. Bounded loops were first discussed in [Culler 85]; the following schema is due to Arvind and Culler [Arvind 86b]:



The governor box can prevent iterations from proceeding by preventing the predicate value from reaching the switches — if there is no predicate value, no tokens can enter the body block. The governor is also informed when an iteration number becomes free by means of the signal tree derived from the outputs of the D-Ks. Our algorithm for providing signals guarantees that when a token is received by each of the D-Ks for a given iteration, then all body instructions, and therefore all predicate instructions and switches, for that iteration have executed. When a token has exited from each D-K, therefore, we conclude that no tokens exist with the previous iteration number, and it can therefore be recycled.

The simplest implementation of the governor box is simply as an arc, initialized with K tokens having different iteration numbers. This allows K iterations to proceed immediately, and additional iterations to begin as tokens arrive from the signal tree[8]. Realistically, we must provide a way to initialize these tokens, and a way to clean them up when the loop finishes. These refinements, however, are complex, not very enlightening, and beyond the scope of this thesis.

We now finish the implementation of loop by including loop constants. As we stated before, we wish to use constant area for holding loop constants. Two instructions manipulate the constant area: constant-store(j) writes location j, and constant-fetch(j) reads it.

——— **Machine Instruction** constant-store(j) ———

Input:	**Outputs:**
⟨DATA, c, i, q, 1, v⟩	⟨DATA, c, i, Dest(q), Pos(q), [Anything]⟩, and v is written into location j of c's constant area

——— **Machine Instruction** constant-fetch(j) ———

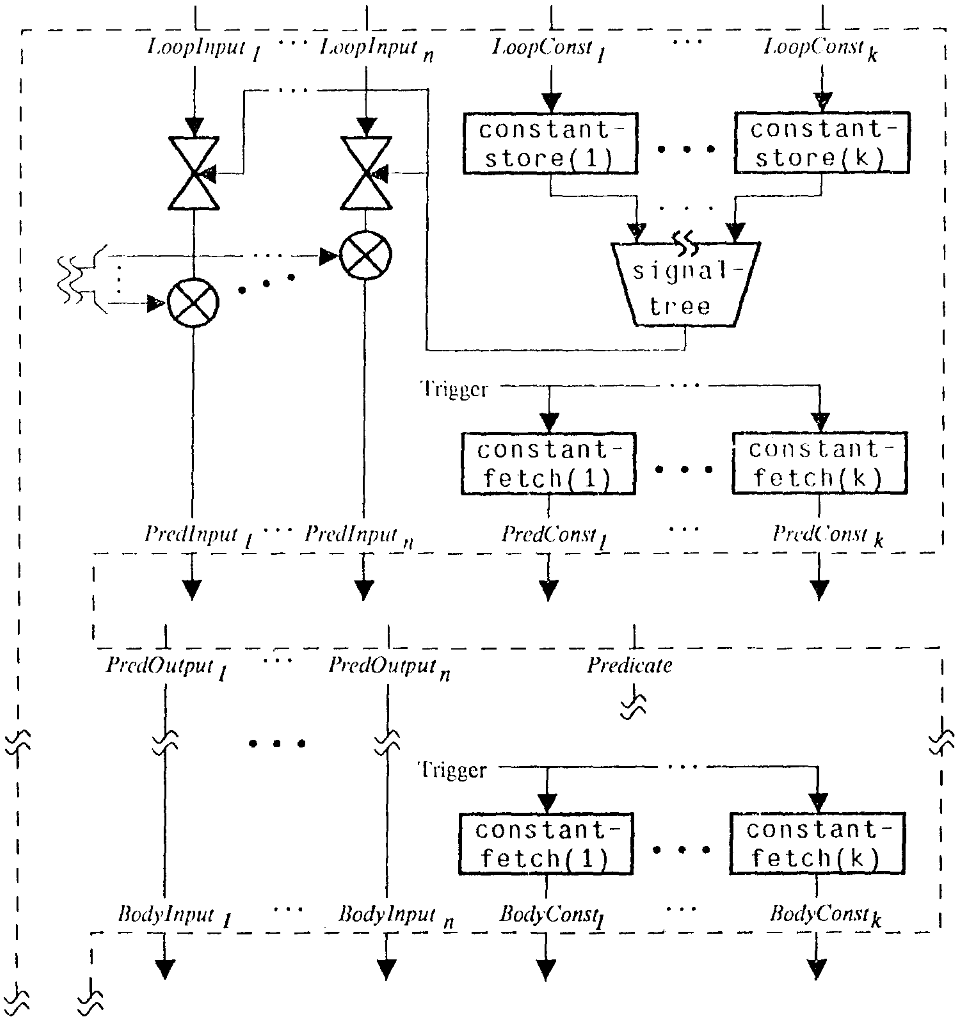
Input:	**Output:**
⟨DATA, c, i, q, 1, v⟩	⟨DATA, c, i, Dest(q), Pos(q), Constant j⟩

The constant-store instruction produces a signal when it has written the indicated location. Because of the proximity of constant area to the ALU, this operation takes no more time than any other instruction. The constant-fetch instruction behaves much like the constant instruction, ignoring its input (trigger) value, except that its output value is taken

[8] Because the predicate for an iteration j has already executed before the governor allows iteration j to proceed, the actual number of tokens preset on the governor arc is $K-1$.

from constant area rather than the instruction itself. In the tagged-token architecture, references to constant area can be pushed into other instructions in the same way that ordinary constants can; again, we will ignore this fact here.

The full translation for `loop` with constants is shown below.



Notice that we prevent the loop from executing until the loop constants have all been stored; this is necessary to prevent premature fetches from constant area. In practice we can usually eliminate a few of these gates by analysis of the predicate. In practice we also push `constant-fetch` instructions through any `if`s contained in the body in order to save switches.

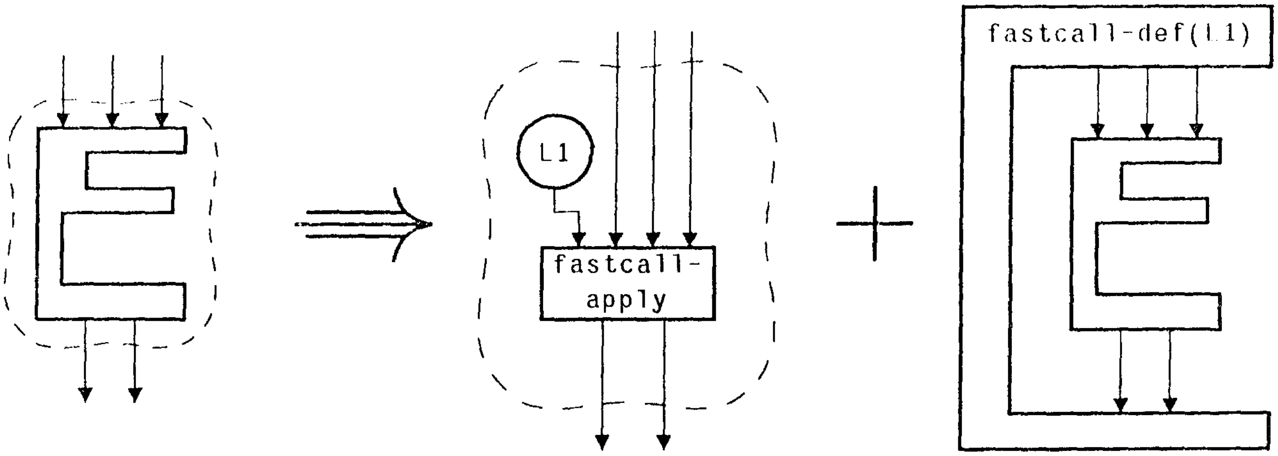
7.5 Switching Contexts

The alert reader will notice that nested loops cannot be accommodated by the translation given in the previous section. The problem is this: suppose we have an outer loop which executes *i* iterations, and its body contains an inner loop which executes *j* iterations each time it is invoked. Each instruction in the body of the inner loop therefore executes a total of *ij* times. How are we to assign unique tags to the iterations of the inner loop? A more serious problem is constant area: the outer loop requires just one constant area, but the inner loop requires *i* constant areas, one for each time it is invoked by the outer loop. The values of the inner loop's constants are likely to vary from one iteration of the outer loop to the next, as they are generally functions of the outer loop's newified variables.

We apparently need to obtain a new constant area each time we begin executing the inner loop, but the outer loop's context maps to only one constant area. Therefore, each execution of the inner loop must take place in a new context. This solves not only the constant area problem, but also the tagging problem, for in each new context we are free to use the iteration field to distinguish the inner loop's iterations. If we had tried to keep the inner loop in the old context we probably would have tried to use the same set of iteration numbers for different instances of the inner loop, leading to clashes since the context fields would also have been the same.

Every context has associated with it a constant area and a code block. As we stated before, each inner loop context requires its own constant area, mutually distinct as well as distinct from the outer loop's constant area. On the other hand, all inner loop contexts execute the same dataflow code, and so they will all share the same code block. The inner loop instructions are disjoint from the surrounding outer loop instructions, however, so we have two code blocks: one for the inner loop and one for what remains of the outer loop along with its surrounding code[9]. This division into two code blocks is illustrated below:

[9] In principle, there is no reason why we could not have them share the same code block, by assigning disjoint sets of offsets. There seems to be no advantage in doing this, though.



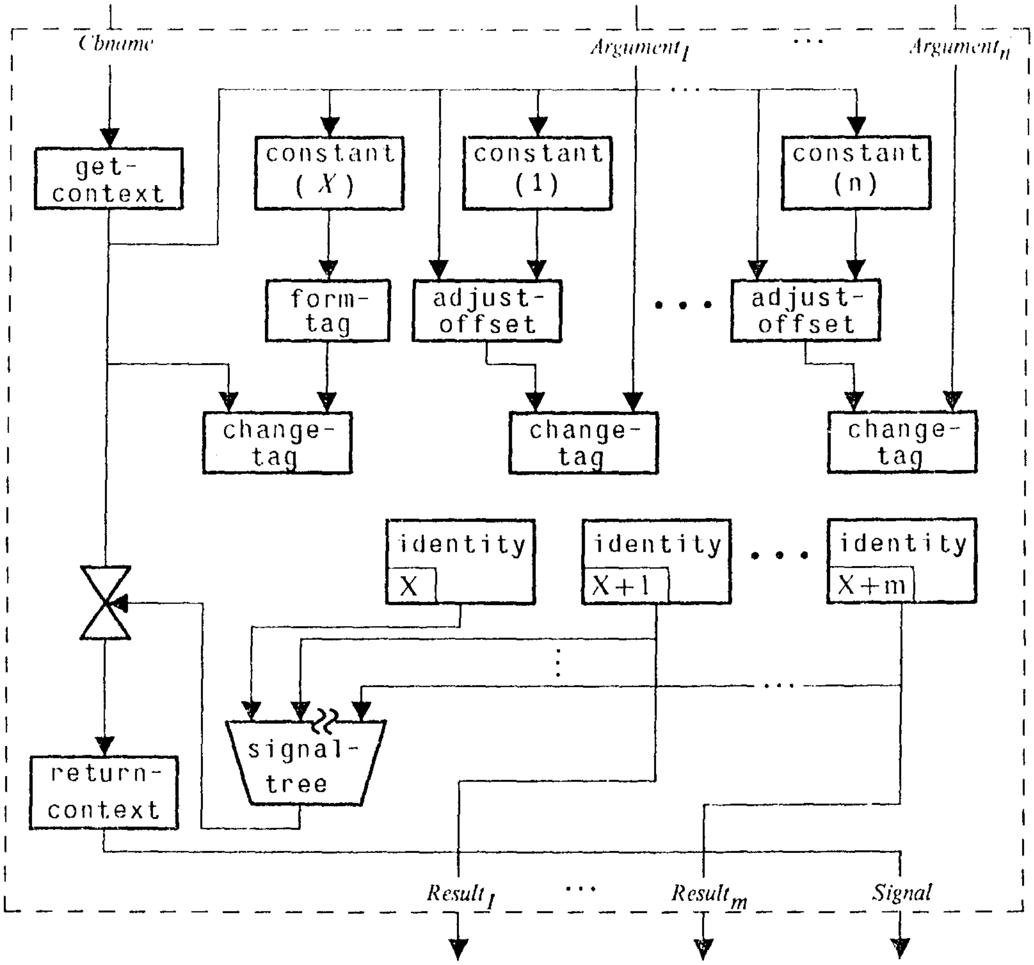
In the figure only the `loop` instruction and its interior has been split into the new code block, but we can also choose to include a small amount of the surrounding code, especially if this reduces the number of arcs crossing the dotted line. Each arc crossing the dotted line represents a token traveling between contexts, this transport being supervised by two new program graph instructions, `fastcall-apply` and `fastcall-def`. `Fastcall-apply` obtains a new context, and sends tokens flowing on arcs entering the split region. It also receives tokens that exit the split region, and deallocates the new context when all of these have been received. `Fastcall-def` performs the complementary function of `fastcall-apply`; it receives the tokens sent by `fastcall-apply` and sends back results.

The names `fastcall-apply` and `fastcall-def` were chosen for these instructions because the process of switching contexts resembles a procedure call. The "arguments" passed are the inputs to the loop, some of which will be stored in constant area as loop constants, and some of which become the initial values for newified variables. The "results" returned are the outputs of the loop; *i.e.*, the final values for the newified variables. (This description might be less accurate if we include some surrounding code in the split region.) Fastcall can be considered as a bare-bones method of obtaining a new context for executing a code block and transporting tokens to and from the new context. The full procedure call mechanism, as described in the next section, uses fastcall as its core.

Possible mechanisms for obtaining and initializing a new context are a topic of current research, but all share a few common characteristics. The object code produced by the compiler has already been partitioned into code blocks, each with a unique name. Each code block carries an indication of how much program memory and constant memory it requires. For example, if a code block contains a loop with five loop constants, then it needs a constant area of size 5. When a code block is to be invoked, the caller supplies the name of the code block, as determined at compile time. The context allocator then finds a free context number, and initializes the context map so that it points to the correct code block and to a fresh constant area of the appropriate size. The allocator returns to the caller a tag containing the new context number, iteration number zero, and the offset of the code block's entry point (usually instruction zero). One possible strategy for allocating contexts is to query a central manager each time a code block is invoked. Another strategy is to have the central manager supply a group of contexts to a procedure invocation, and let the procedure manage these contexts as it chooses for its interior loops. As Arvind and Culler point out [Arvind 86b], we can compute the minimum number of contexts required for deadlock-free execution of a procedure and its loops as a function of the *K* parameters — the number of concurrent iterations — we choose for each loop.

In the schema below we will assume a machine instruction `get-context`, which takes as input a code block name and returns a tag bearing the new context for execution of that code block. Like `allocate-array`, `get-context` sends a token to the manager. The manager obtains a new context and empty constant area, makes the appropriate entries in the context map, and finally sends the tag to the destination of the `get-context` instruction. We will also assume a `return-context` instruction, which informs the manager that it can recycle a context, and returns a signal token indicating that the instruction has executed. Again, it should be remembered that `get-context` and `return-context` might actually be a collection of instructions that manipulates a previously obtained set of contexts, rather than instructions which query the manager.

The implementation of `fastcall-apply` is shown below:



A trigger from the current block causes a constant identifying the called region to be sent to a context allocator. By convention, we send the return address to the called code block's entry point, and the arguments to consecutive locations following the entry point. Sending the arguments is facilitated through the adjust-offset instruction which adds a given number to the offset field of a given tag, and the change-tag instruction which takes a value and a tag and combines the two into a token, effectively sending the value to the context and instruction denoted by the tag.

——— **Machine Instruction** adjust-offset ———

Inputs:	**Output:**
<DATA, c, i, q, 1, <C, I, Q>>	<DATA, c, i, Dest(q), Pos(q), <C, I, Q + v>>
<DATA, c, i, q, 2, v>	

——— **Machine Instruction** `change-tag` ———

Inputs:	**Output:**
<DATA, *c*, *i*, *q*, 1, <*C*, *I*, *Q*>>	<DATA, *C*, *I*, *Q*, 1, *v*>
<DATA, *c*, *i*, *q*, 2, *v*>	

The return address is a tag containing the current context and iteration numbers, and the offset of the first of a set of consecutive `identity` instructions that will receive the results. We create the return address with the aid of the `form-tag` instruction, which combines the offset (computed at compile time) with the current color.

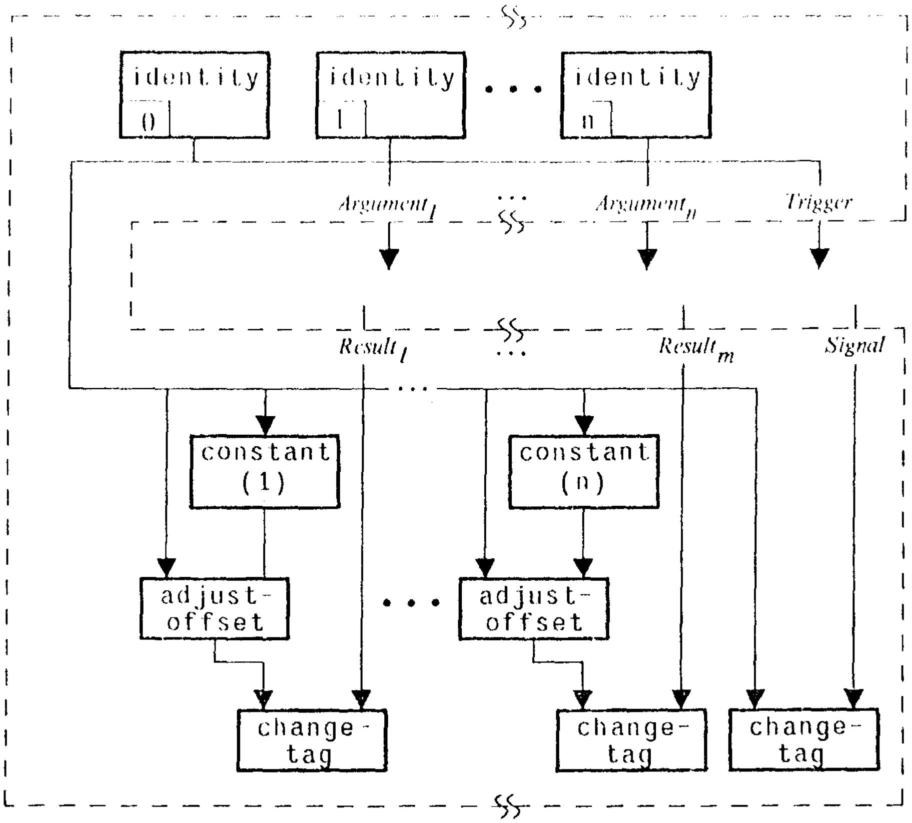
——— **Machine Instruction** `form-tag` ———

Input:	**Output:**
<DATA, *c*, *i*, *q*, 1, *Q*>	<DATA, *c*, *i*, Dest(*q*), Pos(*q*), <*c*, *i*, *Q*>>

By convention, the first of the `identities` will receive a signal token, and any results returned by the called region will arrive on consecutive instructions following this identity. When we receive a token on each of the result identities as well as the signal identity, therefore, we can conclude that all instructions in the called code block have completed execution. By induction, this also implies that any calls the *callee* may have made have also terminated. A signal tree detects the reception of all returned tokens, triggering the release of the context previously obtained. The output of the `return-context` instruction becomes a signal for the current block, not only because the `return-context` would be otherwise unconnected, but also because its execution implies that the `change-tag` instructions have all executed[10].

The implementation of `fastcall-def` is complementary to that of `fastcall-apply`:

[10] This implies that splitting regions must precede generation of signals and triggers.

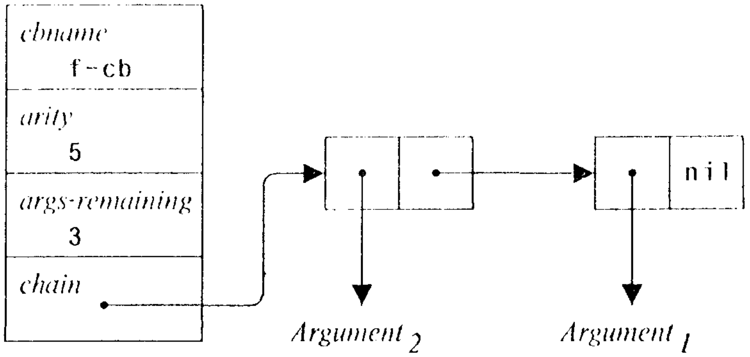


One important feature of the fastcall mechanism is that it is non-strict, in the sense that tokens are transported as soon as they arrive, no matter what their order.

7.6 Procedure Calls

The fastcall mechanism we used to invoke loop "procedures" forms the basis of how we invoke procedures as visible to the Id Nouveau programmer. Id procedures have the additional wrinkle that arguments are accumulated one at a time, because of the curried interpretation of multiple arguments. Before discussing procedure linkage, therefore, we will explain the implementation of closures.

Suppose we have a procedure of five arguments, `f`, and the compiler has named the code block containing `f`'s `def` instruction `f-cb`. Then a closure representing `f` applied to two arguments is represented as:

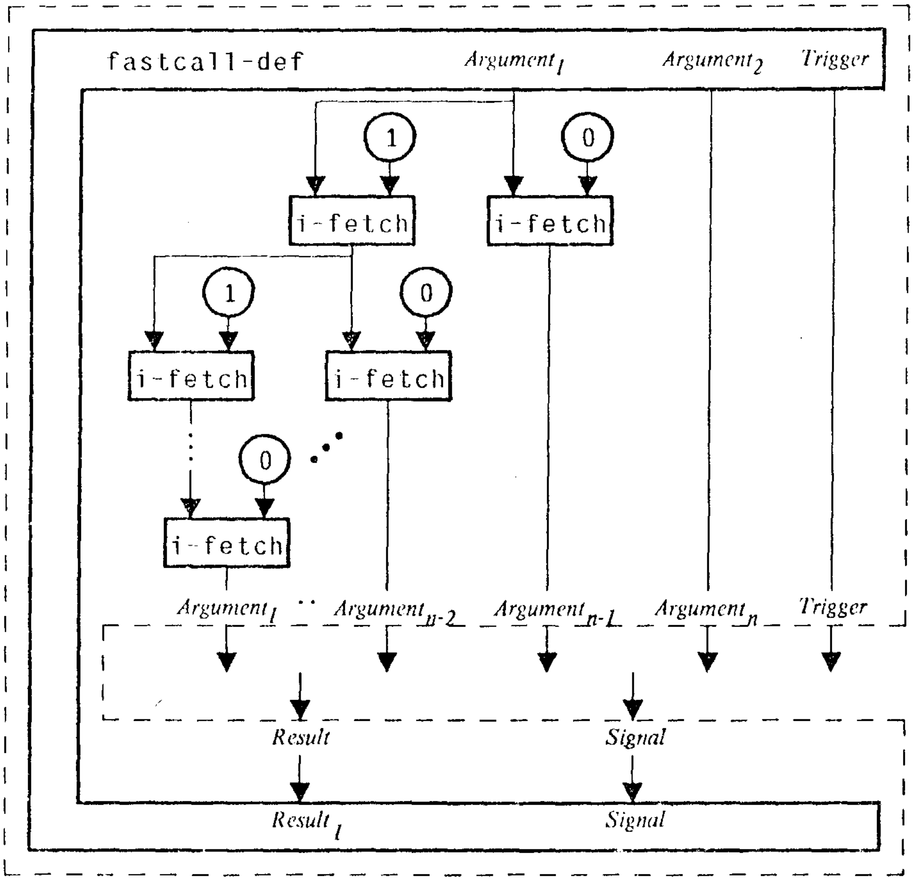


The closure has four fields: a code block name, the arity (5, in the example), the number of arguments not yet specified (5 – 2, or 3, in the example), and a pointer to a linked list of accumulated arguments. We must represent the arguments in a chain because the lower numbered arguments may be shared between different closures. For example, the closure depicted in the figure could be passed to two different `apply` instructions, and the resulting closures would share the first and second arguments but not the third. A procedure with no accumulated arguments, which in the Id program appears as simply the name of the procedure, is represented by a closure whose chain field contains the end-of-chain indicator `nil`.

The implementation of `apply` must first examine the number of remaining arguments required by the incoming closure. If the closure needs only one more argument, then the arity is satisfied, the final argument being the one received by the `apply`. In that case, the `apply` must obtain a context for the execution of the procedure, send the arguments, and receive the results. If, on the other hand, the arity is not yet satisfied, `apply` must allocate a new 2-tuple, add it to the front of the closure's argument chain, and return a new closure with the appropriate components.

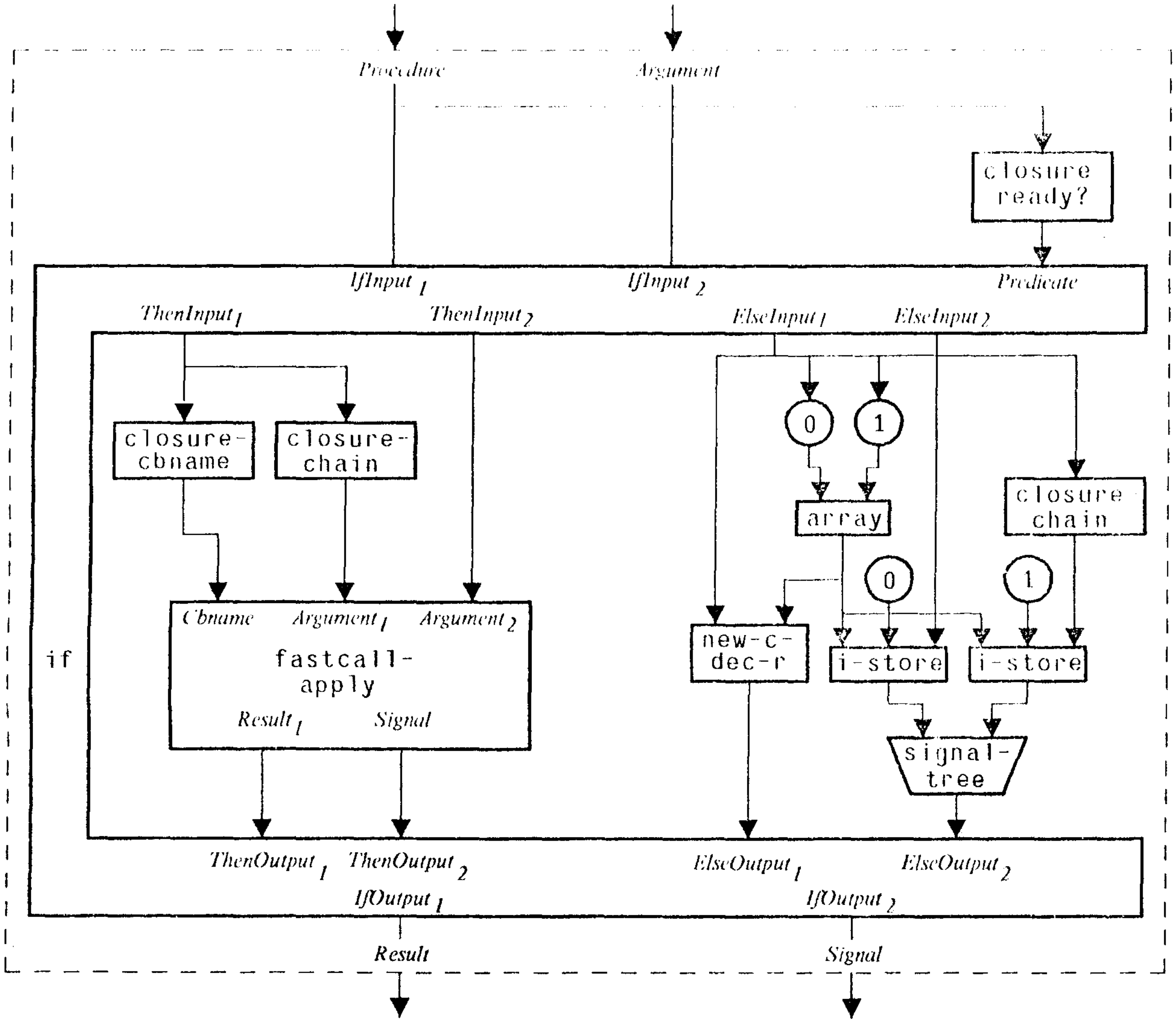
When `apply` actually invokes a procedure, all arguments save the last have been collected in a chain, the final argument being directly available to the `apply`. The simplest approach, therefore, it to have `apply` send the chain along with the final argument directly into the called procedure, which unpacks the chain into individual arguments. The called procedure receives only two tokens (the chain and the final argument), and returns only one token (since Id procedures return only one result).

The mechanism for obtaining the new context and transporting the tokens was already defined in the last section; this is the fastcall mechanism. The implementations of `apply` and `def`, the instructions representing Id procedure linkage, are defined in terms of `fastcall-apply` and `fastcall-def`, the instructions handling context management and token transport. The translation of `def` is shown below.



As the figure shows, `def` is no more than a `fastcall-def`, augmented with some code for unpacking the incoming chain into individual arguments.

The schema for `apply` is a bit more complicated, since it must check the number of arguments remaining for the incoming closure. If more than one, `apply` builds a new closure whose chain contains the new argument. Otherwise, `fastcall-apply` is used to obtain a context and send the chain and final argument to the called procedure.



We've introduced a few instructions for manipulating closures:

——— **Machine Instruction** `closure-cbname` ———

Input: <DATA, c, i, q, 1, <n, a, r, S>>

Output: <DATA, c, i, Dest(q), Pos(q), n>

——— **Machine Instruction** `closure-chain` ———

Input: <DATA, c, i, q, 1, <n, a, r, S>>

Output: <DATA, c, i, Dest(q), Pos(q), S>

——— **Machine Instruction** `closure-ready?` ———

Input:	**Output:**
$\langle \mathrm{DATA}, c, i, q, 1, \langle n, a, r, S\rangle\rangle$	$\langle \mathrm{DATA}, c, i, \mathrm{Dest}(q), \mathrm{Pos}(q),$ `true` if $r = 1$, else `false`$\rangle$

——— **Machine Instruction** `dec-r-new-c` ———

Input:	**Output:**
$\langle \mathrm{DATA}, c, i, q, 1, \langle n, a, r, S_1\rangle\rangle$	$\langle \mathrm{DATA}, c, i, \mathrm{Dest}(q), \mathrm{Pos}(q), \langle n, a, r-1, S_2\rangle\rangle$

Careful consideration will confirm the use of I-structures allows `apply` to create a new closure or invoke a code block even if not all arguments are present; application is therefore non-strict, as desired. When the argument list is unpacked, some fetches may be deferred until arguments arrive.

In closing, we note that sometimes a procedure of known arity is applied to all of its arguments at once. In that case, we want to avoid the overhead of collecting the arguments in a chain, only to have the chain unpacked immediately thereafter. To take advantage of such situations, we need only modify the `apply`, not the `def`. The modified `apply` takes all n arguments at once — n being the arity of the called procedure — and sends them directly into the body of the called procedure, bypassing the unpacking code imbedded in the called procedure's `def`. This modified `apply`, therefore, is exactly like `fastcall-apply`, except that it sends tokens to different offsets within the called procedure. Not having to modify `def` means that the same procedure can be called via both ordinary `apply` and modified `apply` from within the same program.

We now summarize the various linkage mechanisms:

Caller's Schema	*Callee's Schema*	*Used For*	*Features*
`fastcall-apply`	`fastcall-def`	**Internally generated code blocks (e.g. inner loops).**	***N* arguments, *M* results, no I-structures.**
`apply`	`def`	**User defined procedures in general case; closures.**	**1 argument at `apply`, *N* arguments at `def`, 1 result; uses I-structures to collect and pass arguments.**
Variation of `fastcall-apply`	`def`	**User defined procedures whose caller knows arity and applies all arguments at once.**	***N* arguments, 1 result, no I-structures.**

8. Machine Graph Optimizations

Just as a conventional compiler performs optimizations both on the intermediate program form and on the object code, so does our dataflow compiler perform optimizations both on the program graph and on the machine graph. In both cases, the latter optimization phase takes the form of "peephole" optimizations, in which small regions of code are replaced by more efficient equivalents which take advantage of instruction set peculiarities. In a conventional compiler, this entails recognizing sequences of consecutive instructions which fit certain patterns, while for us it means recognizing certain combinations of adjacent nodes in the machine graph. Before looking at what kinds of peephole optimizations are profitable, we describe some aspects of the tagged-token dataflow architecture's instruction set in more detail.

8.1 More Instruction Set Details

In Section 5.1, we were a little vague about how constants are indicated in machine instructions and how the `gate` instruction is implemented. As it turns out, just about all of the peephole optimizations we will discuss take advantage of these two mechanisms. They are described in full detail here.

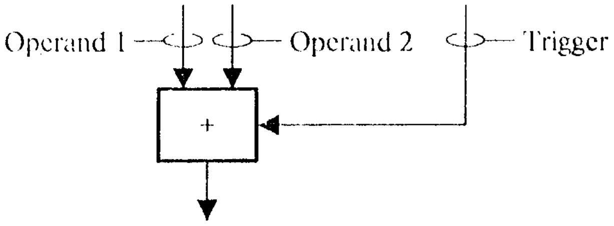
In Section 5.1, it was noted that tokens have a field called *Position*, indicating which operand of a binary instruction a token represents. On the other hand, it was never stated how tokens destined for unary instructions are distinguished from those heading for binary instructions — a necessary distinction, because the former bypass the Waiting-Matching unit. In fact, tokens carry an additional bit called *Partner-P*[11], which indicates whether the token must pass through the waiting-matching unit. A token for a unary instruction, therefore, has this bit set to zero, while both tokens for a binary instruction have this bit set to one.

Having separate *Position* and *Partner-P* fields decouples whether matching takes place from the number of operands an instruction requires. Consider the unary `identity` instruction. This instruction normally receives a token with *Position* set to one (since the identity operation needs only one operand) and the *Partner-P* bit set to zero. Now suppose we send two tokens to this instruction, both with *Partner-P* set to one, and one with *Position* set to

[11] The "P" at the end stems from a Lisp convention wherein predicate functions are suffixed with a "P".

one and the other with *Position* set to zero. By convention, a *Position* field of zero indicates that the data carried by a token is to be ignored. These tokens will match in the waiting-matching unit, and the first will be used as input to the identity. Thus, we have actually implemented the gate instruction!

We can use this trick for any unary instruction, allowing us to obtain synchronization without the need for an explicit gate instruction. We will use the graphical convention that an arc drawn to the side of an instruction is being used as a trigger only, its value being ignored. For example:



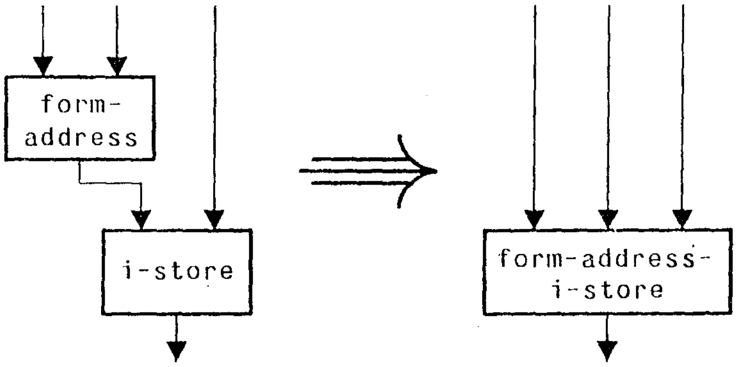
The other aspect of machine instructions glossed over earlier is the inclusion of constants. We stated that there is a special constant(*v*) instruction, but this is not quite the case. In actuality, *any* instruction may have a constant as one of its inputs; an instruction contains two extra fields, one for a constant, and one to indicate for which operand the constant is intended. If the constant position field is zero, then the instruction has no constant. For example, a / instruction whose constant field is 10 and whose constant position field is 2 is effectively a unary instruction which divides its input by 10. The constant field can contain two types of values: a *literal constant*, which gives the actual value to be used, and a ***constant area** pointer*, which gives the offset into constant area from which the constant is to be fetched.

Given the foregoing, it is apparent that the constant(*v*) instruction as it was presented in Section 5.1 is really just an identity instruction, with constant *v*, constant position 1, and whose trigger token has position 0 and partner-p 0. The constant-fetch instruction is implemented in the same way, except that the constant field of the instruction carries a constant area pointer rather than a literal constant. Finally, the constant-store is really a two operand instruction whose second operand indicates which constant is to be stored, this operand always being a constant in the translation schemata described here.

Summarizing, we can say that an instruction can take its inputs from two sources, incoming tokens and constants contained within the instruction, as long as it receives exactly one or two tokens and exactly zero or one constants. This implies that three-input instructions are possible, as long as two inputs come from tokens and one input comes from a constant. We cannot accept three tokens, because the waiting-matching unit only handles pairs. Likewise, we cannot have two constants, because the instruction format accommodates only one. Despite the restrictions, we can often combine a `form-address` and an `i-store` instruction into a single, three-input `form-address-i-store`. Likewise, `adjust-offset` and `change-tag` can almost always be combined into a `adjust-offset-change-tag` instruction.

8.2 Peephole Optimizations

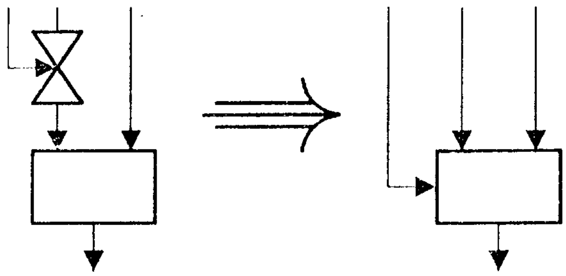
The features of the architecture discussed in the last section provide the opportunity for a variety of useful peephole optimizations. One of the simplest, I-Store Elision, has already been alluded to:



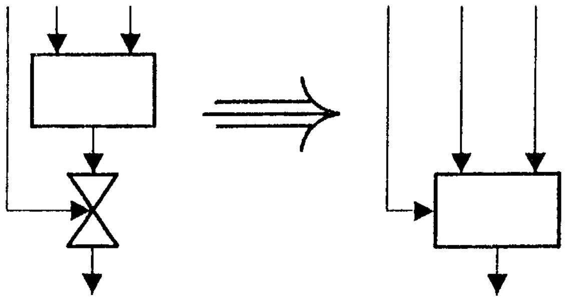
To conform to the machine's constraints, this optimization can only be performed if two of the inputs are tokens and the third is a constant (this is not to say that the constant must be in position three, of course). Experience shows that the optimization is successful in a high percentage of the cases, due to the frequency of stores where the index is a constant, as when a tuple construct is translated, and stores where the structure is a loop constant.

In the remaining descriptions of optimizations, we assume that the optimization is only performed when the final result meets the machine constraints *vis a vis* number of token inputs and number of constant inputs, since these constraints are always present.

Often we can eliminate identity instructions serving the role of gates by taking advantage of the fact that any synchronization can be performed for any instruction. The following transformation is called Trigger Propagation:



In addition to the case shown in the figure, there are analogous symmetrical and unary cases as well. A less obvious variation of this is called Trigger Back-Propagation[12]:

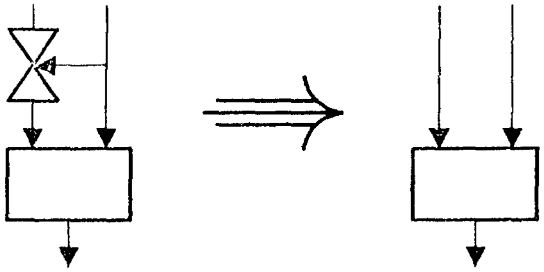


This, too, has an analogous unary case. In addition to the usual machine constraints, Trigger Back-Propagation has the additional constraint that the upper instruction must be side-effect free, since the arrival of the trigger could be contingent on that side-effect's execution. In

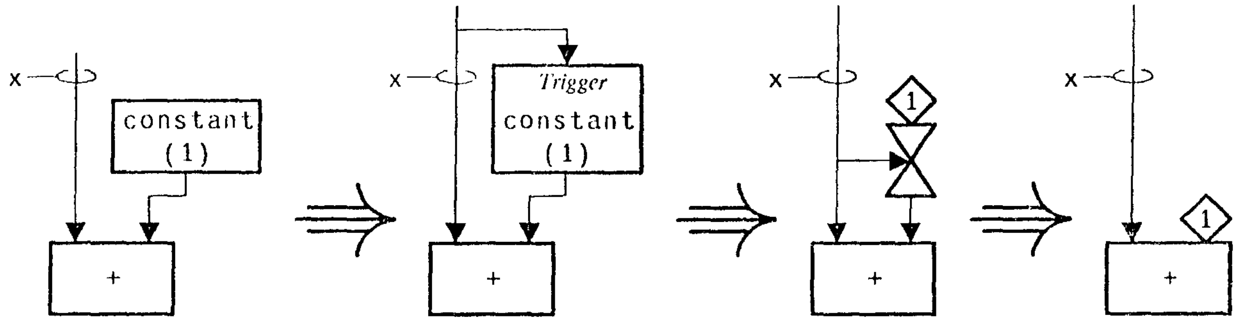
[12] The Trigger Propagation and Trigger Back-Propagation optimizations were suggested by David Culler.

that case, the transformation could result in deadlock. It should also be noted that if the top instruction fed other instructions besides the identity shown, these destinations must still receive their input from the untriggered version of the top instruction. More on this later.

One of the most important peephole optimizations are the Trigger Elimination transformations, one of which is shown below.



This most commonly arises due to expressions in the Id program like x + 1, which would be compiled and optimized as follows:



Program Graph

Well-Connected Program Graph

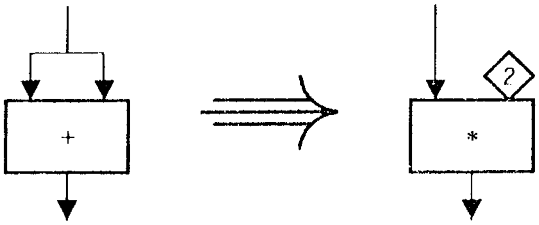
Machine Graph

Optimized Machine Graph

(Again we've introduced some new notation: a value enclosed in a diamond indicates a constant imbedded in a machine instruction.) The triggering shown in the figure is not quite what was described in Section 6.1. The discussion in that section was in fact simplified;

constant instructions which feed arithmetic, i-store, or i-fetch instructions and which have at least one non-constant input are in fact triggered from one of the non-constant inputs to the latter instruction. This triggering strategy is used specifically to expose the opportunity for Trigger Elimination.

The last example of a peephole optimization we consider illustrates that compilation issues for dataflow machines are often quite different than for conventional architectures:



This is exactly the opposite of "reduction in strength" performed by conventional compilers. For them, the left-hand form is more desirable because + is presumably faster than *. In a dataflow machine, the right-hand form is probably more desirable because it reduces the number of tokens passing through the waiting-matching unit by two and the total number of tokens by one.

8.3 A General Peephole Optimization Algorithm

Several attempts to produce specification driven peephole optimizers for conventional compilers have been made. Some rely on attribute grammars [Ganapathi 82], some on other specification techniques [Giegerich 82]. A very simple pattern-directed technique can be used for optimizing machine graphs in a dataflow compiler, one which can handle all of the peephole optimization mentioned in the previous section. As with most dataflow optimizations, the effectiveness as compared to conventional compilers is greatly enhanced by the simplicity of the safety criteria.

Our approach is to specify a peephole optimization by a *pattern*, which specifies a configuration of machine instructions suitable for optimization, and a *replacement*, which gives the more efficient equivalent. A simple functional notation suffices for specifying the pattern and replacement. For example, the Trigger Elimination optimization described in the last section can be specified as

F[identity[A, **triggered by** B], B] → F[A, B]

Here the capital letters, which we will call "pattern variables", represent arbitrary opcodes or arcs, as appropriate. Given a one or more such descriptions, the following algorithm can be used to apply an optimization.

1) Find a collection of instructions that matches the pattern.

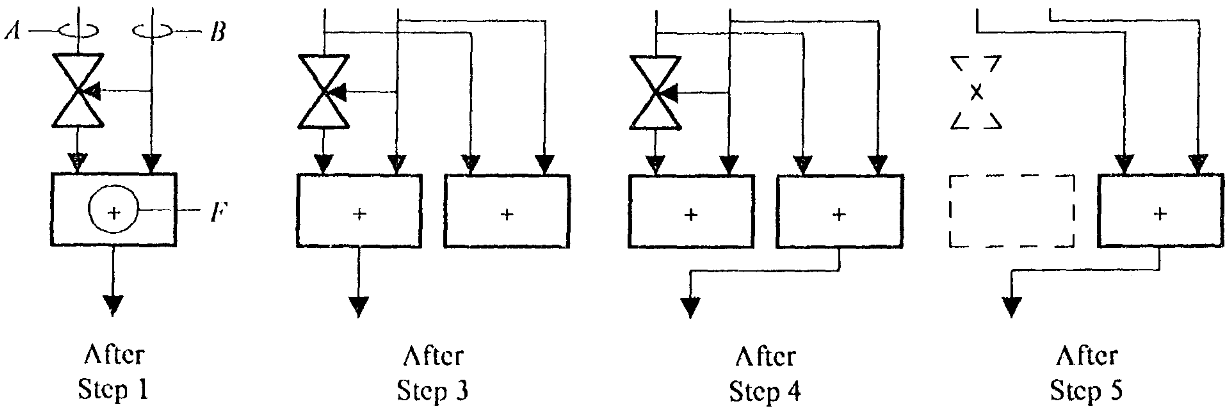
2) Verify that the replacement would meet the machine constraints regarding number of token/constant inputs. If not, give up, otherwise proceed to the next step.

3) Instantiate the replacement, making use of the correspondences between pattern variables and their values as determined in Step 1.

4) Move arcs emanating from the output of the pattern to the output of the replacement.

5) Eliminate dead code. If this step fails, restore the graph to its original state, otherwise the optimization is complete.

These steps are illustrated for Trigger Elimination below.



Step 5, dead code elimination, involves removing instructions that no longer have any arcs emanating from their outputs, the act of which can turn other instructions into dead code. An instruction which causes side-effects is not dead code, however, even if its outputs are unconnected. We cannot eliminate such an instruction, but we cannot leave it unconnected either, for that would destroy our ability to detect the termination of the instruction's code block. Dead code elimination fails if we reach such a situation, and we must surrender our intention to carry out the optimization.

The following algorithm is used to systematically attempt all optimizations on an entire code block.

1) Initialize a list of candidates to a list of all instructions in the code block.

2) If the candidate list is empty, we are finished. Otherwise, remove an instruction from the candidate list, call it *x*.

3) Select an optimization from the list of all optimizations, and try to apply it to *x*, where *x* is to be matched with the output of the optimization's pattern. The optimization may fail because *x* does not match the pattern, because the replacement would violate machine constraints, or because dead code elimination failed. If the optimization succeeded, proceed to Step 4, otherwise try another optimization, going back to Step 2 when all optimizations have been tried.

4) Add to the candidate list all instructions created when the replacement was instantiated, as well as all instruction's connected to the replacement's output. Go back to Step 2.

Step 4 is needed because the application of an optimization can create new opportunities for optimization. We have not specified the order in which optimizations should be considered in Step 3, nor have we given a rule for selecting an instruction in Step 2. Unfortunately, varying these can affect the numbers of optimizations successfully performed. Currently under investigation are ways of determining and/or specifying the most advantageous orderings.

9. Conclusions

A general purpose programming language, Id Nouveau, was presented, in the form of a syntactic-sugar-free subset Id Kernel. We then gave schemata for the translation of Id Kernel into program graphs, an abstract sort of dataflow graph in which all control and data flow are encoded as data dependencies. The program graph form is reminiscent of Livermore's IF1 format [Skedzielewski 85b]. The program graph was also considered as an abstract intermediate program form that serves as a framework for program optimization. After examining the MIT Tagged-Token Dataflow Architecture, we then completed the compilation process by describing the translation from program graph to well-connected program graph, and from there to machine graph, or object code. Finally, machine-dependent peephole optimizations on the machine graph were discussed.

The author has recently completed a compiler based on the concepts presented here. The compiler implements all of the program graph and machine graph schemata presented here, and includes a pattern-directed peephole optimizer as described in Chapter 8. Results from the peephole optimizer are quite encouraging: it reduced the static code size of a 1000 line program by about 20%. Few of the program graph optimizations have been put in place, the exception being constant folding. Implementation of other program graph optimizations is expected within a year.

There are many directions for future work in this area. Certainly the topic of compiler optimizations can be explored in greater depth, and algorithms and heuristics can be developed for the detection of opportunities for code motion. All optimization algorithms will benefit from a more thorough inquiry into intra- and inter-procedural analysis, particularly side-effect, strictness, and data type analysis.

One very exciting avenue leads from the firing rule description of program graphs. In [Nikhil 86], an operational semantics is given for Id Nouveau in terms of rewrite rules. This semantics can be taken as a definition of the language. On the other hand, the firing rules given for program graph instructions together with the translation schemata form the beginnings of another kind of operational semantics. The machine code produced by the compiler can be readily verified against the latter semantics, since all that needs to be done is verify that the collection of machine graph operators implementing a particular program graph instruction satisfy its firing rule. Given that, it is reasonable to pursue a proof that the rewrite semantics and the firing rule semantics are consistent. If such a proof is found, then we have the first known provably correct dataflow implementation of an I-structure language, and probably the first provably correct dataflow implementation of *any* programming language.

Finally, the non-sequential aspect of I-structure languages leads one to consider how one can compile efficient code for conventional, sequential architectures from these languages. This not a contradiction; the non-sequential nature of I-structure languages does not preclude sequential implementation, but only demands that a sequential implementation simulate parallelism to some degree. To what degree a simulation is necessary, or in other words what is the maximum sequential thread size allowable, is a topic of research. This work would have significance not merely for sequential architectures, but also for any architecture that attempts to combine the parallelism of dataflow machines while avoiding the extremely fine grain of parallelism those machines support.

References

[Ackerman 84] W. B. Ackerman.
Efficient Implementation of Applicative Languages.
Technical Report TR-323, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, April, 1984.

[Aho 86] A. V. Aho, R. Sethi, and J. D. Ullman.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley, Reading MA, 1986.

[Allen 72] F. E. Allen and J. Cocke.
A Catalogue of Optimizing Transformations.
In R. Rustin (editor), *Design and Optimization of Compilers*, pages 1-30. Prentice-Hall, Englewood Cliffs NJ, 1972.

[Arsac 82] J. Arsac and Y. Kodratoff.
Some Techniques for Recursion Removal from Recursive Functions.
ACM Transactions on Programming Languages and Systems 4(2):295-322, April, 1982.

[Arvind 78] Arvind, K. P. Gostelow, and W. Plouffe.
An Asynchronous Programming Language and Computing Machine.
Technical Report 114, University of California, Irvine, Department of Information and Computer Science, Irvine CA, December, 1978.

[Arvind 83] Arvind and R. A. Iannucci.
Instruction Set Definition for a Tagged-Token Data Flow Machine.
Computation Structures Group Memo 212-3, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, February, 1983.

[Arvind 84] Arvind and J. D. Brock.
Resource Managers in Functional Programming.
Journal of Parallel and Distributed Computing 1:5-21, 1984.

[Arvind 85] Arvind and R. A. Iannucci.
Two Fundamental Issues in Multiprocessing: The Dataflow Solution.
Computation Structures Group Memo 226-3, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, August, 1985.

[Arvind 86a] Arvind, K. Pingali, and R. S. Nikhil.
I-Structures: Data Structures for Parallel Machines.
Computation Structures Group Memo, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, 1986.
(In Preparation).

[Arvind 86b] Arvind and D. E. Culler.
Managing Resources in a Parallel Machine.
In *Fifth Generation Computer Architectures 1986*, pages 103-121. Elsevier Science Publishers B.V., 1986.

[Culler 85] D. E. Culler.
Resource Management for the Tagged Token Dataflow Architecture.
Technical Report TR-332, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, January, 1985.

[Ferrante 83] J. Ferrante and K. J. Ottenstein.
A Program Form Based on Data Dependency in Predicate Regions.
In *Conference Record of the 9th ACM Symposium on the Principles of Programming Languages*, pages 217-236. Association for Computing Machinery, January, 1983.

[Ganapathi 82] M. Ganapathi and C. N. Fischer.
Description-Driven Code Generation Using Attribute Grammars.
In *Conference Record of the 9th ACM Symposium on the Principles of Programming Languages*, pages 108-119. Association for Computing Machinery, January, 1982.

[Giegerich 82] R. Giegerich.
Automatic Generation of Machine Specific Code Optimizers.
In *Conference Record of the 9th ACM Symposium on the Principles of Programming Languages*, pages 75-81. Association for Computing Machinery, January, 1982.

[Gurd 85] J. R. Gurd, C. C. Kirkham, and I. Watson.
The Manchester Prototype Dataflow Computer.
Communications of the ACM 28(1):34-52, January, 1985.

[Hiraki 84] K. Hiraki, T. Shimada, and K. Nishida.
A Hardware Design of the Sigma-1, A Data Flow Computer for Scientific Computations.
In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 524-531. IEEE Computer Society, August, 1984.

[Johnsson 85] T. Johnsson.
Lambda Lifting.
In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 201)*, pages 190-203. Springer-Verlag, Berlin, September, 1985.

[Nikhil 86] R. S. Nikhil, K. Pingali, and Arvind.
Id Nouveau.
Computation Structures Group Memo 265, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, July, 1986.

[Pingali 86] K. Pingali.
Private Communication.
1986.

[Skedzielewski 85a]
S. K. Skedzielewski and M. L. Welcome.
Data Flow Graph Optimization in IF1.
In *Functional Programming Languages and Computer Architectures (Lecture Notes in Computer Science; 201)*, pages 17-34. Springer-Verlag, Berlin, September, 1985.

[Skedzielewski 85b]
S. K. Skedzielewski and J. R. W. Glauert.
IF1, an Intermediate Form for Applicative Languages.
Reference Manual M-170, Lawrence Livermore National Laboratory, Livermore CA, July, 1985.

[Steele 78] G. L. Steele.
RABBIT: A Compiler for SCHEME.
Technical Report AI-TR-474, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge MA, May, 1978.

[Wetherell 82] C. S. Wetherell.
Error Data Values in the Data-Flow Language VAL.
ACM Transactions on Programming Languages and Systems 4(2):226-238, April, 1982.